SCALABLE FORMAL DYNAMIC VERIFICATION OF MPI PROGRAMS THROUGH DISTRIBUTED CAUSALITY TRACKING

by

Anh Vo

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2011

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Anh Vo**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan	, Chair	**04/26/2010** Date Approved
Robert M. Kirby	, Member	**04/18/2010** Date Approved
Bronis R. de Supinski	, Member	**04/18/2010** Date Approved
Mary Hall	, Member	**04/18/2010** Date Approved
Matthew Might	, Member	**04/18/2010** Date Approved

and by **Al Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Almost all high performance computing applications are written in MPI, which will continue to be the case for at least the next several years. Given the huge and growing importance of MPI, and the size and sophistication of MPI codes, scalable and incisive MPI debugging tools are essential. Existing MPI debugging tools have, despite their strengths, many glaring deficiencies, especially when it comes to debugging under the presence of nondeterminism related bugs, which are bugs that do not always show up during testing. These bugs usually become manifest when the systems are ported to different platforms for production runs.

This dissertation focuses on the problem of developing scalable dynamic verification tools for MPI programs that can provide a coverage guarantee over the space of MPI nondeterminism. That is, the tools should be able to detect different outcomes of nondeterministic events in an MPI program and enforce all those different outcomes through repeated executions of the program with the same test harness.

We propose to achieve the coverage guarantee by introducing efficient distributed causality tracking protocols that are based on the *matches-before* order. The matches-before order is introduced to address the shortcomings of the Lamport *happens-before* order [40], which is not sufficient to capture causality for MPI program executions due to the complexity of the MPI semantics. The two protocols we propose are the Lazy Lamport Clocks Protocol (LLCP) and the Lazy Vector Clocks Protocol (LVCP). LLCP provides good scalability with a small possibility of missing potential outcomes of nondeterministic events while LVCP provides full coverage guarantee with a scalability tradeoff. In practice, we show through our experiments that LLCP provides the same coverage as LVCP.

This thesis makes the following contributions:

- The MPI matches-before order that captures the causality between MPI events in an MPI execution.
- Two distributed causality tracking protocols for MPI programs that rely on the matches-before order.
- A Distributed Analyzer for MPI programs (DAMPI), which implements the two

aforementioned protocols to provide scalable and modular dynamic verification for MPI programs.

- Scalability enhancement through algorithmic improvements for ISP, a dynamic verifier for MPI programs.

For my parents

"The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong, it usually turns out to be impossible to get at or repair"

– Douglas Adams, *Mostly Harmless*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My research that led to this dissertation and this dissertation itself would not have been possible without the advice and support from my friends, my family, the faculty of the School of Computing, and our research collaborators. First and foremost, I would like to thank my advisor, Professor Ganesh Gopalakrishnan, who I consider to be the best advisor one could ever hope to have. His level of enthusiasm has inspired many students, myself included, to be self-motivated and work hard to achieve the goals we have set for ourselves. Thank you, Ganesh; I could not have done it without your support.

The dissertation would not have reached its current state without the suggestions of Professor Robert M. Kirby, who is my coadvisor, Dr. Bronis de Supinski, Professor Mary Hall, and Professor Matthew Might, who are my committee members. Thank you all of you for your valuable input on my research and my dissertation.

During the course of my research on DAMPI, I probably maintained an average ratio of about five bad ideas to one decent idea, and about five decent ideas to one good (i.e., publishable) idea. Without the countless brainstorming sessions with our research collaborators, Bronis de Supinski, Martin Schulz, and Greg Bronevetsky, whether at the whiteboard or through emails, the bad ones would have been considered good, and vice versa. Working with you has been an eye opening experience. I am especially grateful to Bronis for his editing help with the papers, the proposal, and this dissertation itself.

My research began as a project to improve ISP, a dynamic code level model checker for MPI. ISP is the work of Sarvani Vakkalanka, whom I had the pleasure of working with for two years. She was one of the smartest and hardest working colleagues that I have had. Her work on ISP inspired my research and I am thankful for that. I would also like to express my gratitude to many of my colleagues, especially Michael Delisi, Alan Humphrey, Subodh Sharma, and Sriram Aananthakrishnan, for their input and contributions.

With a few exceptions, going to graduate school is usually demanding and stressful. I would not have made it without the support of my family and my friends. My parents, my sister, and my cousin and her family have always been my biggest supporters. I appreciate

that they always try to tactfully ask me how many years I have been in school instead of asking me how many years I have left. I am especially indebted to my fiance, Phuong Pham, for providing me the motivation and the encouragement I needed to finish the dissertation work. I am also thankful to many good friends who have made my graduate school experience memorable. My-phuong Pham, Linh Ha, Hoa Nguyen, Huong Nguyen, Khiem Nguyen, Thao Nguyen, Huy Vo, and Trang Pham, thank you all.

Five years ago as I was pondering the decision on whether I should continue my studying, it was my friend Khoi Nguyen who encouraged me to go for graduate school. And until today I still remember the countless nights we stayed up together, each working on our own projects and paper, even though we are several time zones apart. Khoi will always have my gratitude.

CHAPTER 1

INTRODUCTION

It is undeniable that the era of parallel computing has dawned on us, regardless of whether we are ready or not. In a recent report entitled *The Future of Computing Performance: Game Over or Next Level*, the National Research Council states that "the rapid advances in information technology that drive many sectors of the U.S. economy could stall unless the nation aggressively pursues fundamental research and development of parallel computing" [25]. Today supercomputers are becoming faster, cheaper, and more popular. The last release of the Top500 list in November 2010 witnessed five new supercomputers, which had not previously been on the list before, making it to the top ten [16]. Future growth in computing power will have to come from parallelism, from both the hardware side and the software side. Programmers who are used to thinking and coding sequential software now have to turn to parallel software to achieve the desired performance. Unfortunately, the transition from sequential programming to parallel programming remains a challenge due to the complexity of parallel programming.

There are many different forms of parallelism in software, from multithreading to message passing. This dissertation specifically focuses on message passing software, especially those written in MPI (Message Passing Interface [28]). Today MPI is the most widely used programming API for writing parallel programs that run on large clusters. The ubiquity of MPI can be attributed to its design goals, which are flexibility, performance, and portability. MPI accomplishes these goals by providing a very rich semantics that incorporates the features of both asynchronous and synchronous systems. Synchronous behavior is easy to use and understand, which allows developers to achieve higher productivity, while asynchronous behavior allows for the highest performance. Both of these properties are necessary for a ubiquitous standard.

Unfortunately, the performance and flexibility of MPI come with several debugging challenges. MPI programs, especially under the presence of nondeterminism, are notoriously hard to debug. Nondeterminism bugs are difficult to catch because repeated unit

testing, which is the most commonly used method to test concurrent code, usually covers only a small number of possible executions [73]. When the code enters production and is deployed in different environments, the untested (buggy) path becomes manifest and might cause the software to crash.

To highlight how difficult debugging can get with MPI, we consider a simple MPI program shown in Figure 1.1, which contains a very subtle bug. In this program, the asynchronous nondeterministic receive posted in process P_1 can potentially match with message sent by either P_0 or P_2. Under traditional testing, one may never successfully be able to force P_2's message (which triggers `ERROR`) to match. While this option appears impossible due to its issuance after an MPI barrier, it is indeed a possible match because the MPI semantics allows a nonblocking call to pend until its corresponding wait is posted. This example illustrates the need for more powerful verification techniques than ordinary random testing on a cluster where, due to the absolute delays, P_2's match may never happen (and yet, it may show up when the code is ported to a different machine).

Even though there are many techniques and tools that help developers discover MPI nondeterminism errors, they basically fall into one of these three categories: static methods, dynamic methods, and model checking. Static methods have the advantages of being input-independent since they verify the program at the source code level. However, they tend to provide too many false alarms, especially for a large code base, due to the lack of runtime knowledge. Model checking methods are very powerful for small programs in terms of verification coverage but they quickly become impractical for large software due to the infeasibility of building models for such software. Dynamic methods such as testing or dynamic verification are the most applicable methods for large MPI programs since they produce no false alarms and also require little work from the tool users. This dissertation focuses on applying formal techniques to create efficient and scalable dynamic verification tools for MPI programs.

P_0	P_1	P_2
`Isend(to` P_1`,22)`	`Irecv(from:*,x)`	`Barrier`
`Barrier`	`Barrier`	`Isend(to` P_1`,33)`
`Wait()`	`Recv(from:*,y)`	`Wait()`
	`if(x==33) ERROR`	
	`Wait()`	

Figure 1.1: MPI example to illustrate POE

1.1 Dynamic Verification for MPI

Most realistic MPI programs are written in Fortran/C/C++ and run on clusters with thousands to hundreds of thousands of cores. These programs can have not only the common C/C++/Fortran bugs such as memory leaks or buffer overflow, but also bugs specific to MPI such as deadlocks or illegal buffer reuse. Earlier we presented a buggy example involving a nondeterministic receive, which is troublesome for developers to debug because the bugs appear intermittently and do not show up in all traces. Testing tools for MPI such as Marmot [35] and Umpire [68] are unreliable for such bugs because they only catch the bugs that appear in the testing run. In other words, they do not provide any coverage guarantee over the space of nondeterminism.

The model checker MPI-SPIN [58] can provide a coverage guarantee for MPI verification. However, MPI-SPIN requires the users to build models manually using the SPIN programming language for MPI programs and run the model checker to verify the models. For realistic MPI programs containing hundreds of thousands of lines of code, this requirement is unrealistic and renders this approach impractical.

While there exist dynamic verification tools for other types of parallel software such as CHESS [46] or Verisoft [26], similar tools for MPI are still nonexistent.

1.2 Thesis Statement

Scalable, modular and usable dynamic verification of realistic MPI programs is novel and feasible.

1.3 Contributions

1.3.1 MPI Matches-Before

We investigate the Lamport *happens-before* [40] order between events in a distributed system and show that it is insufficient for capturing the full semantics of MPI executions. More specifically, the reason is that the happens-before order relies on knowing when an event finishes all its execution effects. However, obtaining such information for MPI events is a challenging task since an MPI event can exist in many different states from the point in time when the process invokes the MPI call to the point where the call no longer has any effect on the local state. We show that either the point of issuing or the point of completion is insufficient to order events in an MPI execution correctly, which is counterintuitive to what most tool developers tend to think. To overcome these limitations, we contribute the notion of *matches-before* which focuses on the matching

point of MPI events (intuitively, the matching point is the point when an operation *commits* that it will finish according to the commitment).

1.3.2 Lazy Update Protocols

We introduce two fully distributed protocols, namely the Lazy Lamport Clocks Protocol (LLCP) and the Lazy Vector Clocks Protocol (LVCP). Both of these protocols rely on the matches-before order to track causality between nondeterministic events in MPI executions. While the vector clock-based protocol provides a complete coverage guarantee, it does not scale as well as the Lamport clock-based protocol. We show through our experiments that in practice, the Lamport clock protocol provides the same coverage guarantee without sacrificing scalability.

1.3.3 Contributions to ISP

ISP is a formal dynamic verifier for MPI programs developed originally by Sarvani Vakkalanka [64–66, 71]. ISP uses a centralized version of matches-before to achieve verification coverage over the space of nondeterminism. My specific contributions to ISP are as follows:

- Studying the scalability of ISP and making ISP scale to handle realistic MPI applications through various algorithmic improvements such as reducing ISP memory footprint through data structure improvements, increasing speed up through the use of better communication mechanisms, and parallelization of the ISP scheduler with OpenMP [66, 67, 71, 72].
- Interfacing with GEM [34] developers to make ISP a practical tool.

1.3.4 DAMPI

The lazy update algorithms provide the basis for developing scalable and portable correctness checking tools for MPI programs. We demonstrate this by providing the implementation for these algorithms through a new tool called DAMPI [69, 70], which is a **D**istributed **A**nalyzer for **MPI** programs. Similarly to ISP, DAMPI's goals are to verify MPI programs for common errors such as deadlocks, resource leaks over the space of nondeterminism. In contrast with ISP, DAMPI is fully distributed and targets large scale MPI programs that run on large clusters. The lazy update algorithms allow DAMPI to provide coverage over the space of nondeterminism without sacrificing scalability. Further, we implement several user configurable search bounding heuristics in DAMPI such as loop

iteration abstraction, which allows the user to specify the regions for which DAMPI should bypass during the verification, and bounded mixing, which is a mechanism that allows the user to limit the impact a nondeterministic choice has on subsequent choices. Both of these heuristics aim to reduce the search space and provide the user with configurable coverage.

CHAPTER 2

BACKGROUND

This chapter gives the background knowledge about causality tracking in distributed systems in general, and MPI in particular.

2.1 Distributed Systems

While there are several possible ways to define what distributed systems are, we adapt the definition from Coulouris [22], which defines a distributed system as *a collection of networked computers that communicate with each other through message passing only.* Since we mostly restrict our study to the software level, we find the concept of distributed programs more useful and applicable. *A distributed program P is a collection of processes $P_0, \ldots, P_n$ communicating through message passing, running a common program to achieve a common goal.* It is important to note that this definition allows a distributed program to run even on a single computer where each process P_i runs within its own virtual address space provided by the host operating system. In the rest of the paper, we shall use the term distributed system in place of distributed program.

2.2 Distributed Causality Tracking

The ordering of events is an essential part of our daily life. Consider the following hypothetical example: Bob receives two undated letters from his father; one of which says "Mom is sick" and the other says "Mom is well." Since the letters are undated, Bob has no way to reason about the current well-being of his mother. One apparent solution is for Bob to pick up the phone and call his father to inquire about his mother's status. However, let us assume that in this hypothetical time and space, telephone communication does not exist, which would also explain why Bob's father sent him letters instead of calling. With this constraint, one possible solution is for Bob's father to write down the time from his wristwatch to the end of each letter. In other words, he is attaching the *physical clock* to each message that he sends to Bob.

This solution works fine if Bob's father is the only one communicating with Bob. It is not hard to imagine why this scheme would fail if another person, e.g., Bob's sister, also communicates with Bob. Assuming that instead of receiving two letters from his father, Bob receives one from his father that says "Mom is sick" and one from his sister that says "Mom is well." If Bob's sister uses her own wristwatch to timestamp her message to Bob and her watch is not in synchronization with his father's watch, the scheme still does not allow Bob to order the events properly based on the received messages. He would not be able to figure out whether his sister received a message from his father updating the status of the mother (and told her to send a message to Bob) after his father had sent him the message, or she simply visited the mother before she became ill. In other words, the scheme does not fully capture the causal relation of the two messages.

In distributed systems, causality tracking is a major part of many problems, ranging from the simplest problems of resource allocation and reservation to more complicated problems such as checkpointing or deterministic replay. Many of these algorithms are used in safety critical systems and faulty knowledge would have catastrophic consequences. We now look at several ways that one can track causality in distributed systems and how they can help Bob solve the problem of figuring out his Mom's current health status.

2.2.1 Lamport Clocks

In 1978, Lamport invented a very simple yet effective mechanism to capture the total order of events in distributed systems [40]. Instead of using physical clocks, process P_i now has a function $C_i(a)$ that returns a number $C(a)$ for event a in P_i, and we shall call this number a's timestamp (or a's clock). In other words, instead of associating physical times to P_i's events, the algorithm now associates *logical times* to them.

Assuming that sending and receiving messages are observable events in the system and that local events follow program order, we describe the Lamport clocks algorithm through a set of clock maintenance rules as follows:

- $R1$. Each process P_i maintains a counter C_i initialized to 0.
- $R2$. P_i increments C_i when event e occurs and associates e with the new clock. Let $C_i(e)$ denote this value.
- $R3$. P_i attaches (piggybacks) C_i whenever it sends a message m to P_j
- $R4$. When P_i receives m, it sets C_i greater than or equal to its present value and greater than the clock it receives.

Figure 2.1 shows a message passing program with three processes implementing the above Lamport Clock algorithm. Each event has an associated clock value and the direction of the arrow indicates the direction of the message (i.e., a, c, e are the sending events and b, d, f are the corresponding receiving events, respectively).

The above algorithm has two important properties:

- $P1$. If event a occurs before event b in P_i, then $C_i(a) < C_i(b)$. This follows from rule $R2$ above.
- $P2$. If a is the sending event of message m and b is the corresponding receiving event of m, then $C_i(a) < C_i(b)$. This follows from rule $R3$ and $R4$ above.

We are now ready to define the Lamport *happens-before* ($\rightarrow$) relation for the set of all events in a distributed system. Let e_i^a be the a^{th} event that occurs in process P_i, $send(P_i, m)$ be the event corresponding to sending message m to P_i, and $recv(P_i)$ be the event corresponding to the reception of m from P_i, $\rightarrow$ is defined as follows:

$$e_i^a \rightarrow e_j^b \Leftrightarrow \begin{cases} (i = j \wedge a + 1 = b) \vee \\ (i \neq j \wedge (e_i^a = send(P_j, m) \wedge e_j^b = recv(P_i, m))) \vee \\ (\exists e_k^c : e_i^a \rightarrow e_k^c \wedge e_k^c \rightarrow e_j^b) \end{cases}$$

Using this definition of $\rightarrow$ and applying the two properties $P1$ and $P2$, we can see that any distributed system implementing the Lamport clocks algorithm satisfies the *Clock Condition*, which states: for any two events a and b, if $a \rightarrow b$ then $C(a) < C(b)$. It is important to note that the converse of the clock condition is not always true. Consider events e and c in Figure 2.1, while $C(e) < C(c)$, we cannot conclude that $e \rightarrow c$. However, we can infer that c could not have happened before e. While this inference is enough for

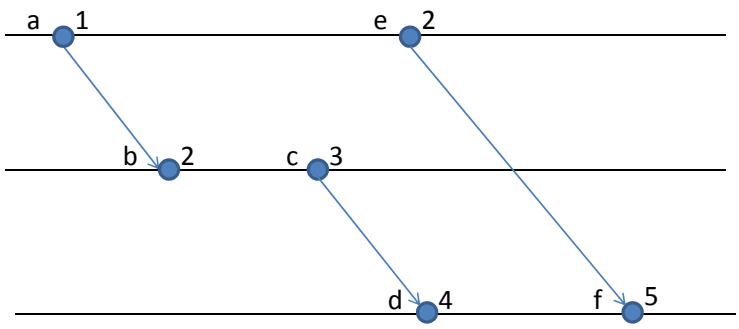

Figure 2.1: A distributed system using Lamport clocks

several applications, some of them do require a more meaningful answer (i.e., whether e happens-before c, or e and c are simply *concurrent* events). We will now look at vector clocks, a more powerful scheme of logical clocks that can address the aforementioned deficiency of Lamport clocks.

2.2.2 Vector Clocks

Vector clocks have been used a long time before they are formally defined simultaneously and independently by Fidge [24] and Mattern [43]. For example, version vectors, which are essentially vector clocks, were used to detect mutual inconsistency in distributed systems [49].

Vector clocks address the limitation of Lamport clocks by maintaining a vector of timestamps per process. That is, process P_i maintains a vector $VC_i[0..n]$ where $VC_i[j]$ represents P_i's knowledge about the current timestamp of P_j. We now describe the vector clocks algorithms:

- $R1$. Each process P_i has a vector VC_i initialized to 0 ($\forall k \in \{0..n\} : VC[k] = 0$).
- $R2$. P_i increments $VC_i[i]$ when event e occurs and assigns e the new clock. Let $e.VC$ denote this value.
- $R3$. P_i attaches (piggybacks) V_i whenever it sends a message m to P_j. Let $m.VC$ denote this value.
- $R4$. When P_i receives m, it updates its vector clock as follows: $\forall k \in \{0..n\} VC_i[k] = max(VC_i[k], m.VC[k])$.

We also need to define a way to compare vector clocks (which is not necessary for Lamport clocks since we are only dealing with a single integer). Two vector clocks VC_i and VC_j are compared as follows (we only show the case for $<$, the $=$ case is trivial and thus omitted, the $>$ case is similar to that of $<$):

$$VC_i < VC_j \Leftrightarrow \forall k \in \{0..n\} : VC_i[k] \leq VC_j[k] \wedge \exists l \in \{0..n\} : VC_i[l] < VC_j[l]$$

Earlier we mentioned the fact that the Lamport clocks algorithm cannot guarantee the converse of the *Clock Condition*. The vector clocks algorithm effectively addresses that deficiency, which means it satisfies the *Strong Clock Condition*: for any events a and b: $a \rightarrow b \; iff \; a.VC < b.VC$ (in contrast with Lamport clocks, which only guarantee that if $a \rightarrow b$ then $a.LC < b.LC$)

Figure 2.2 shows the same parallel program as Figure 2.1 using vector clocks instead of Lamport clocks. Now consider events e and c, which have vector timestamps of $[2, 0, 0]$

and $[1, 2, 0]$, respectively. Apparently, neither $e \rightarrow c$ nor $c \rightarrow e$ holds. In this case, e and c are *concurrent* events.

Definition 2.1 Two events a and b are *c*oncurrent if $a \nrightarrow b \wedge b \nrightarrow a$

While vector clocks are useful in applications that require the knowledge of the events' partial order, they have one major drawback: each message has to carry a vector of n integers. As systems scale beyond thousands of processes, the impact on bandwidth becomes significant. Unfortunately, under the worst case scenarios, the size limitation of vector clocks is a necessary requirement [20]. Nonetheless, in systems where bandwidth is a large concern, one can apply several compression schemes to reduce the size of the vector clocks that are transmitted [31, 44, 60, 62]. The effectiveness of these schemes are highly dependent on the communications pattern and also on the properties of the communicating channels.

2.3 The Message Passing Interface

The Message Passing Interface (MPI) is a message-passing library interface specification [28], designed to help programmers write high performance, scalable, and portable parallel message passing programs. Today it is the *de-facto* API for writing programs running on large clusters. A description of an MPI program can be found in the MPI standard [28], which states:

> An MPI program consists of autonomous processes, executing their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations

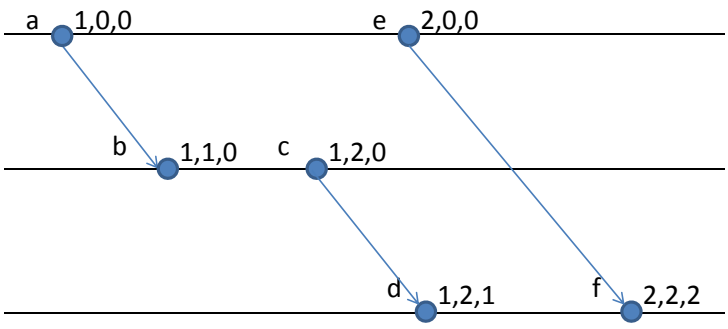

Figure 2.2: A distributed system using Lamport clocks

> of MPI are possible. This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified.

A typical MPI program is written in C/C++/Fortran, compiled, and linked with an MPI implementation. There exist many different MPI implementations [9, 11, 29], all of which follow the specifications given in the standard.

An example of a typical MPI program is given in Figure 2.3. The program tries to compute π as follows: each process tries to compute its own chunk based on the numerical integration method, using the number of intervals it receives from the master through `MPI_Bcast`, which is a broadcasting call. The master would then collect the chunks to calculate the final results of π through `MPI_Reduce`, which is a reduction call.

To provide the maximum performance and portability, MPI supports a wide range of communication modes including nonblocking communication, nondeterministic receives, and a large number of collective calls. We will divide these communication calls into three groups, namely asynchronous communication, synchronous communication, and collective communication and describe them in Sections 2.3.1, 2.3.2, and 2.3.3.

2.3.1 Synchronous Point-to-Point Communication

The most basic form of MPI point-to-point communication is through the use of synchronous communication. These calls usually implement some rendezvous protocol where the receiver blocks until it starts to receive data from a matching sender. In the case of a synchronous send, the sender blocks until it receives an acknowledgement from the receiver that is has started the receiving process.

Synchronous communication offers several advantages. First, it is easier to use and understand compared to asynchronous communication, which allows for higher productivity. Second, it can help prevent memory exhaustion by not requiring the MPI runtime to provide message buffer. However, synchronous communication usually comes with performance penalty due to the cost of synchronization, especially for applications that communicate large messages infrequently. In order to address this problem, MPI offers two alternatives: buffered communication and asynchronous communication.

Buffered communication allows the process to issue a sending request and continue processing without waiting for the acknowledgement from the receiver. MPI programmers can take advantage of buffered communication through one of these two methods:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
            if (myid == 0)
                printf("pi is approximately %.16f, Error is %.16f\n",
                       pi, fabs(pi - PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}
```

Figure 2.3: An MPI program calculating π

- Allocate an explicit buffer and provide it to the `MPI_Bsend` call through the use of `MPI_Buffer_attach`. Note that the user can only attach one buffer per process and the buffer can be used for more than one message. `MPI_Buffer_detach` can be called later to force the delivery of all messages currently in the buffer.
- Take advantage of the MPI runtime's buffer through the use of `MPI_Send`. The user buffer is available immediately after the call since MPI has copied the data into its own buffer. However, it is generally unsafe to rely on the runtime to provide such a buffer. In fact, the MPI standard does not mandate that the implementation should provide any buffer (although most of them do in practice to improve performance). If the runtime runs out of buffer space due to excessive pending communication, `MPI_Send` will block until more buffer space is available, or until the data has been transmitted to the receiver side's buffer (normally it only blocks until the user buffer has been completely copied to the runtime's buffer).

We now describe the syntax of `MPI_Ssend` and `MPI_Recv`, which are the two main synchronous point-to-point operations.

- `MPI_Ssend(buffer,count,dtype,dest,tag,comm)`

  where `buffer` is a pointer to the data to be sent, `dtype` is the abstract type of the data, count is the number of elements of type `dtype` in `buffer`, `dest` is the destination process for this message, `tag` is an integer tag associated with this message, and `comm` is the MPI communicator in which this event will take place (a communicator is basically a group of processes created by the program; the special `MPI_COMM_WORLD` is the default communicator for all processes). Note that while `MPI_Send` has the same blocking behavior as `MPI_Ssend`, according to the MPI standard, its behavior is asynchronous. That is, the call can return before a matching receive is posted.
- `MPI_Recv(buffer,count,dtype,source,tag,comm,status)`

  where `buffer` is a pointer to the receiving buffer, `dtype` is the abstract type of the data, count is the number of elements of type `dtype` expected to receive, `source` is the process that is expected to deliver the message, `tag` is the integer tag associated with the expected message, `comm` is the MPI communicator in which this event will take place, and `status` is a data structure that can be used to get more information about the received message. The receive is not required to fill the buffer (i.e., partial receives are allowed), in which case the user can find out exactly how many elements

were received by calling `MPI_Get_count` or `MPI_Get_elements`. Note that it is an MPI error to post a receive that does not have enough buffer space to receive an incoming message.

2.3.2 Asynchronous Point-to-Point Communication

As mentioned earlier, synchronous communication offers robustness and predictability of message delivery at the cost of program flexibility and performance. Many applications exhibit a large degree of communication-computation overlap and thus would benefit from having the ability to issue some communication requests, continue with local processing, and process the results of those requests when the computation phase is over, with the hope that the MPI runtime has sent/delivered the message during the computation phase.

MPI offers asynchronous point-to-point communication through the use of calls such as `MPI_Isend` and `MPI_Irecv`. The process would provide a buffer, issue the call, obtain a request handle from the runtime, and wait for the communication request to finish later using either `MPI_Wait` or `MPI_Test` (or their variants such as `MPI_Waitall` or `MPI_Waitany`). In the MPI 1.1 standard, the process cannot access the buffer while the requests are still pending. This was later changed to read-only for pending sending requests and no-access for pending receiving requests for MPI 2.2 and higher.

Similarly to `MPI_Send`, the MPI runtime can (and often will) buffer the messages sent by `MPI_Isend` as long as the runtime's buffer has enough space. In other words, the corresponding call to `MPI_Wait` simply indicates that the user data have been copied to the runtime's buffer and the process can now reuse the buffer associated with the `MPI_Isend`. Those applications that require a rendezvous semantics for such situations will have to use `MPI_Issend` where the corresponding `MPI_Wait` will block until the receiver has started to receive the data.

We now describe the syntax of `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`.

- `MPI_Isend(buffer,count,dtype,dest,tag,comm,req_handle)`
  where `req_handle` represents the communication handle returned by the MPI runtime. All other arguments are similar to those of `MPI_Send`.
- `MPI_Recv(buffer,count,dtype,source,tag,comm,req_handle)`
  where `req_handle` represents the communication handle returned by the MPI runtime. All other arguments are similar to those of `MPI_Recv`.
- `MPI_Wait(req_handle,status)`
  where `req_handle` is the communication request to be finished and `status` is where

the user can obtain more information about the communication request after it finishes. Note that `req_handle` is set to `MPI_REQUEST_NULL` once the communication associated with this request completes. Invoking `MPI_Wait` on `MPI_REQUEST_NULL` causes no effect.

2.3.3 Collective Communication

As the name suggests, collective communication refers to MPI functions that require the participation of all processes within a defined communicator. It is easy to think of collective communications as a set of point-to-point operations; for example, the `MPI_Bcast` call can be decomposed into multiple `MPI_Send` calls from the root to all other processes in the communicator and multiple `MPI_Recv` calls from the other processes to receive the data from the root. In practice, however, collective operations are heavily optimized by most implementations depending on the size of the messages and the network structure. For example, the `MPI_Bcast` call can use a tree-based algorithm to broadcast the message efficiently [61].

While it is intuitive for developers to consider collective operations as having synchronizing behavior, the implementation is often not required to provide such semantics. There are only a few collective calls that have synchronization semantics such as `MPI_Barrier` while the rest are only required to block until they have fulfilled their roles in the collective operation. For example, in an `MPI_Reduce` call, after a process has sent out its data to the reducing root, it can proceed locally without having to wait for the root to receive all messages from other processes. However, the MPI standard does require that all processes in the communicator execute the collective. Collective operations also have additional requirements such as the sending/receiving buffers have to be precisely specified (i.e., no partial receives allowed).

We now describe the syntax of the `MPI_Barrier` call.

- `MPI_Barrier`(comm)

  where `comm` is the MPI communicator on which this process wants to invoke the barrier call. The MPI standard requires that all processes in the communicator participate in the barrier and that they all block until all processes have reached this routine.

2.3.4 Nondeterminism in MPI

The MPI standard also allows some MPI calls to have nondeterministic behavior to provide programmers with more flexibility and reduce coding complexity. There are several nondeterministic constructs in MPI:

- Nondeterministic receives using `MPI_ANY_SOURCE` as their argument for the `source` field. As the name suggests, these receives can accept any incoming messages carrying a compatible tag and coming from senders within the same communicator. We sometimes refer to nondeterministic receives as wildcard receives.
- Nondeterministic receives using `MPI_ANY_TAG`. In addition to `MPI_ANY_SOURCE`, a receive call in MPI can also choose to accept messages carrying any tag (within the same communicator and coming from a matching sender). A nondeterministic receive can use both `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, in which case it can accept any incoming message from senders belonging to the same communicator. It is also important to note that the communicator cannot be nondeterministic.
- The `MPI_Waitany` call can complete any one of the request handles passed in as its argument (the choice of this request can be arbitrary). Similarly, the `MPI_Waitsome` can complete any number of requests out of all request handles passed in as its arguments (i.e., if there are n request handles to complete, there are $2^n - 1$ possible ways for `MPI_Waitsome` to finish). Note that due to their highly nondeterministic behavior, `MPI_Waitany` is only occasionally used and `MPI_Waitsome` is almost never used.
- The `MPI_Startall` call starts all *persistent requests*, which are communication handles that can be reused over and over again until they are explicitly deallocated, in any arbitrary order and different ordering might lead to different execution paths. However, in practice, most MPI implementations start them in the order given by the array of request handles.
- `MPI_Test` and its variants `MPI_Testany`, `MPI_Testall`, `MPI_Testsome` return whether some pending communication requests have finished or not. If the pending requests have finished, the MPI runtime deallocates the requests and set the flags. Since communication completion depends not only on the order that the requests are issued, but also on network routing, timing, and numerous system factors, the flag returned by `MPI_Test` is not guaranteed to be set at the same time between multiple program executions with the same test harness. For example, during the

first run, the developer might observe that `MPI_Test` sets the flag to true after five invocations; yet during the next run for the same test harness, it sets the flag to true after the seventh invocation. The only thing the MPI standard guarantees is that if the process repeatedly invokes `MPI_Test` in a busy-wait loop, the flag eventually will be set, if both the receiver and the sender have already started the receiving/sending calls (this is called the MPI *progress-guarantee*). Many large programs use `MPI_Test` in place of `MPI_Wait` due to its nonblocking characteristic. The program can periodically check whether some pending communication requests have finished without having to block.

- Nondeterministic probes (`MPI_Probe` or `MPI_Iprobe`) using either `MPI_ANY_SOURCE`, `MPI_ANY_TAG`, or both. Probes allow the process to check whether there are any messages to receive without actually receiving the messages. In applications where the receivers do not always know how large the incoming messages are, probes are extremely useful. If there are ready-to-receive messages, the status field returned by the probe allows the process to determine the exact size of the incoming messages, and thus the process can now allocate just enough buffer to receive them. `MPI_Probe` behaves similarly to `MPI_Recv` in the sense that it blocks until there are messages to receive. In contrast, `MPI_Iprobe` behaves similarly to `MPI_Test`, which returns immediately and sets the flag to true if there are messages to receive. As in the case with `MPI_Test`, `MPI_Iprobe` also has progress-guarantee semantics. It is important to note that if a program invokes a probe call with `MPI_ANY_SOURCE` and later issues a receive with `MPI_ANY_SOURCE`, there is no guarantee that the receive would receive the message probed earlier (unless there is only one possible message to receive).

2.3.5 Nonovertaking in MPI

The rich features and the enormous flexibility of MPI come with the cost of increased complexity. In a program with asynchronous sends/receives interacting with synchronous calls with some or all of them being nondeterministic, trying to determine which sending event should match with each receiving event can be a challenging task. To facilitate the matching of sends and receives, the MPI standard enforces the nonovertaking rule, which states:

> Messages are nonovertaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two

receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

Intuitively, one can imagine the communication universe in MPI being split into multiple FIFO channels. Two processes exchanging messages using the same tag within the same communicator effectively are utilizing one of these FIFO channels. However, the relative order of two messages from two different channels can be arbitrary. We will provide a formal notion for this rule in Chapter 3.

2.3.6 Common MPI Errors

We provide several examples that illustrate the most common errors found in MPI programs. They can be classified in these categories: deadlocks, resource leaks, erroneous buffer reuse, and type mismatches. Some bugs can be caused by the use of MPI nondeterministic constructs as explained earlier in Section 2.3.4. That is, when a bug is caused by nondeterminism, there are MPI program schedules that may not be executed under conventional testing. We try to present a mixture of both nondeterministic bugs and deterministic bugs through several examples.

2.3.6.1 Deadlock

Deadlock typically happens when there is a send and receive mismatch. That is, one process tries to receive a message from a process that either has no intention to or is not able to send the expected message. Figure 2.4 presents a simple program where each process sends a message to P_0, and P_0 tries to receive from all other processes. However, due to a programming bug, P_0's first receive call is expecting a message from P_0 (itself), which does not post any send to match that receive. Therefore, the execution deadlocks.

Figure 2.5 presents an unsafe program involving two processes sending messages to each other (a *head-to-head deadlock*). The deadlock occurs when the size of the buffer exceeds the amount of buffering the MPI runtime provides. The MPI standard recommends against relying on the runtime buffer to achieve the program's objective

```
if (rank != 0)
   MPI_Send(sendbuf, count, MPI_INT, 0, 0, MPI_COMM_WORLD);
else
   for (i = 0; i < proc_count; i++)
      MPI_Recv(recvbuf+i, count, MPI_INT, i, 0, MPI_COMM_WORLD, status+i);
```

Figure 2.4: Deadlock due to send receive mismatch

```
if (rank == 0) {
   MPI_Isend(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
   MPI_Irecv(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
else if (rank == 1) {
   MPI_Isend(buf, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
   MPI_Irecv(buf, count, MPI_INT, 0, 0, MPI_COMM_WORLD, &h);
   MPI_Wait(&h, &status);
}
```

Figure 2.5: Head-to-head deadlock

since it limits program portability. It is unsafe because the MPI standard guarantees that it will either deadlock or execute correctly. This communication pattern exists in the Memory Aware Data Redistribution Engine (MADRE) [59], which is a collection of memory aware parallel redistribution algorithms addressing the problem of efficiently moving data blocks across nodes, and many others.

Figure 2.6 presents an example of an unsafe program in which two MPI processes post `MPI_Barrier` and `MPI_Bcast` calls in a way that could potentially cause a deadlock. Since the MPI standard does not require the implementations to provide synchronizing semantics for `MPI_Bcast`, it is possible (and likely in practice) that P_0 does not have to wait for P_1 to post the corresponding `MPI_Bcast` call before P_0 can finish its `MPI_Bcast` call, which means that the execution does not deadlock. However, if an implementation assumes synchronizing behavior for `MPI_Bcast`, the execution deadlocks. This example again shows that semantic deadlock need not imply observed deadlock.

Figure 2.7 presents a program that contains a nondeterminism deadlock. In this example, P_1 posts two receives, one of which is a wildcard while the other one is specifically

```
if (rank == 0)
   MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Barrier(MPI_COMM_WORLD);
else if (rank == 1)
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(buffer, count, MPI_INT, 0, MPI_COMM_WORLD);
```

Figure 2.6: Deadlock due to collective posting order

```
if (rank == 0) {
   MPI_Send(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1) {
   MPI_Recv(buf, count, MPI_INT, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD, &status);
   MPI_Recv(buf, count, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
else if (rank == 2) {
   MPI_Send(buf, count, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

Figure 2.7: Deadlock due to nondeterministic receive

matches a message from P_2. However, since either send from P_0 or P_2 is eligible to match the wildcard receive, the second receive from P_1 will not have a matching send if the wildcard receive matches the send from P_2. During testing, a developer might observe that the program runs fine during some executions and deadlocks during others.

2.3.6.2 Resource Leaks

There are many different types of resource leaks such as unfreed communicators, unfreed types, and unfreed requests. We provide an example of a request leak in Figure 2.8. In this program, the nonblocking request from P_0 remains in the system since the program never deletes it through a call to `MPI_Cancel`, nor does it wait or test for the request's completion through a call to `MPI_Wait` or `MPI_Test`. Request leaks are a serious issue for MPI programs as an excessive number of pending requests drastically degrades the performance of the program or may crash a long running application.

In addition to having a request leak, this example also contains a different kind of resource leaks: type leak. Both P_0 and P_1 fail to free `newtype` through a call to `MPI_Type_free`. Imagine a program where this pattern is enclosed in a loop that creates many different new MPI datatypes without freeing them; the resources associated with the types never get freed and returned to the system, which in the long run might affect the program's performance or behavior (due to out of memory errors).

2.3.6.3 Erroneous Buffer Reuse

The MPI standard requires that the buffer associated with a nonblocking request not be accessed by the process until the request has been waited or tested for completion.

```
if (rank == 0) {
   MPI_Datatype new_type;
   MPI_Type_contiguous(count, MPI_INT, &newtype);
   MPI_Type_commit(&newtype);
   MPI_Isend(buf, 1, newtype, 1, 0, MPI_COMM_WORLD, &h);
   ...
   MPI_Finalize();
}

else if (rank == 1) {
   MPI_Datatype new_type;
   MPI_Type_contiguous(count, MPI_INT, &newtype);
   MPI_Type_commit(&newtype);
   MPI_Recv(buf, 1, newtype, 0, 0, MPI_COMM_WORLD, &status);
   MPI_Finalize();
}
```

Figure 2.8: Resource leak due to unfreed request

Since MPI 2.2, this requirement is relaxed for nonblocking send operations with respect to read access. That is, the process can read a buffer of a nonblocking send request before the request completes (writing to the buffer is still prohibited). Violating this requirement leads to undefined behavior. The program shown in Figure 2.9 presents a situation of illegal buffer reuse in both the sender side and receiver side.

2.3.6.4 Type Mismatches

MPI's requirements for type matching between sending and receiving are very complex because the standard supports many different methods to create new datatype. The

```
if (rank == 0) {
   MPI_Isend(buf, 1, newtype, 1, 0, MPI_COMM_WORLD, &h);
   buf = 1; /* illegal write to buffer before send request completes */
   MPI_Wait(&h, &status);
}
else if (rank == 1) {
   MPI_Irecv(buf, 1, newtype, 0, 0, MPI_COMM_WORLD, &h);
   a = buf; /* illegal read from buffer before read request completes */
   MPI_Wait(&h, &status);
}
```

Figure 2.9: Erroneous buffer reuse

flexibility limits the ability of the MPI runtime to perform strict type checking and thus many erroneous type mismatches go uncaught during testing yet surface during production runs. Figure 2.10 shows a program that should run correctly in most cases but will produce an erroneous result when running in an environment where the two nodes have different endianness.

2.3.7 The MPI Profiling Interface

Being an API used heavily in high performance computing where the users tend to have strong interest in performing various analyses such as performance measurement and data tracing, the MPI standard defines a profiling interface to facilitate such tasks. The user can take advantage of the profiling interface by providing wrappers for those MPI calls that they are interested in profiling (e.g., `MPI_Send`). The wrapper would then invoke the MPI calls from the runtime by issuing the corresponding PMPI calls (e.g., `PMPI_Send`). Figure 2.11 shows a simple user wrapper that counts the number of times `MPI_Send` is invoked.

The major drawback of the profiling interface provided by the standard is that there can be at most one active wrapper linked with the program. The P^{N}MPI framework [53] allows multiple MPI wrappers to be stacked on top of MPI programs.

2.4 Piggybacking in MPI

Piggybacking is the act of sending additional data (piggyback data) along with messages originated from the main application. Many distributed protocols and algorithms rely on piggybacking support. For example, tracing libraries [39, 55], critical path anal-

```
if (rank == 0) {
   int data = 5;
   /* sending 4 bytes */
   MPI_Send(&data, 4, MPI_BYTE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1) {
   int data;
   /* receive one int */
   MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

Figure 2.10: Type mismatch between sending and receiving

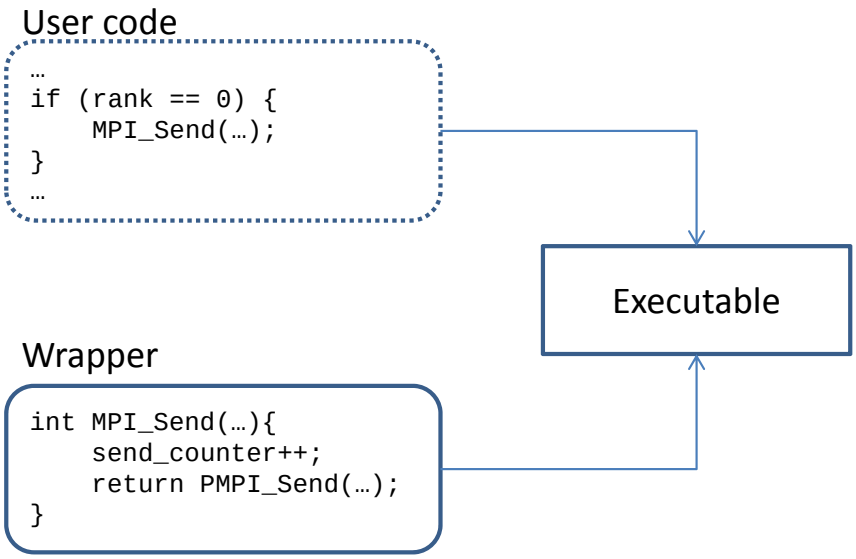

Figure 2.11: A simple PMPI wrapper counting `MPI_Send`

ysis [18], and application-level checkpointing protocol [52] all require piggyback data to function correctly. In addition, causality tracking protocols such as Lamport clocks and vector clocks that we mentioned earlier also require piggyback.

Unfortunately, the MPI standard, as of version 2.2, does not have any built-in piggyback mechanism. Most tools have relied on ad hoc implementations to support piggybacking. We describe here several popular mechanisms of sending piggyback data, each with its own advantages and disadvantages.

2.4.1 Buffer Attachment Piggybacking

Buffer attachment piggybacking, also called explicit packing piggybacking [51], is one of the simplest approaches of piggybacking where the tool layer attaches the piggyback data directly to the message buffer. This scheme involves using `MPI_Pack` at the sender side to pack the piggyback data together with the message data and using `MPI_Unpack` at the receiver side to separate the piggyback data from the main message. The piggyback buffer can be attached at the beginning or at the end of the buffer. Figure 2.12 illustrates the concept of buffer attachment piggybacking.

While this method is simple, it incurs very high overhead, especially in communication-intensive programs, due to the excessive calls to `MPI_Pack` and `MPI_Unpack`. It is also

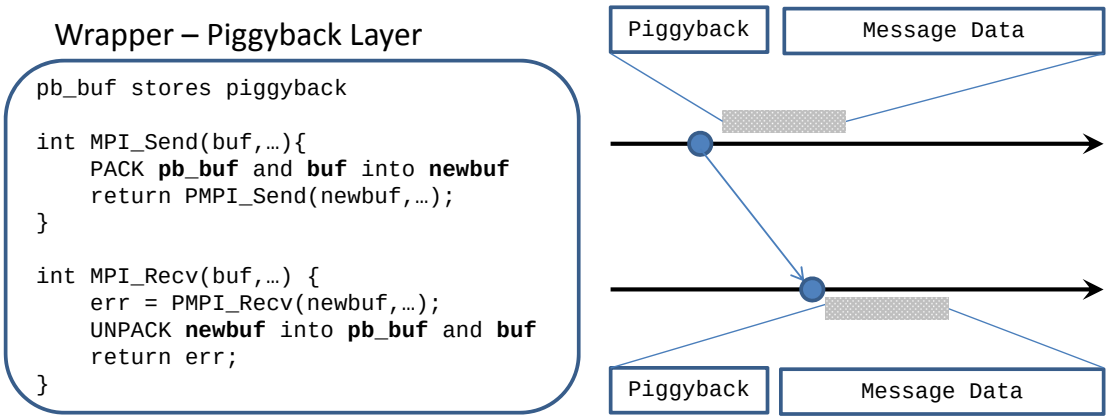

Figure 2.12: Buffer attachment piggyback

not entirely clear how one would attach piggyback data to collective operations such as `MPI_Reduce`. Studies have also shown that this method of piggybacking has the highest overhead in terms of bandwidth and latency [51]. There are currently several MPI tools that use buffer attachment piggybacking [48].

2.4.2 Separate Message Piggybacking

As the name implies, this piggyback scheme involves sending the piggyback data as a separate message, either right before or right after the message originated by the main application. Figure 2.13 illustrates the concept.

The piggyback layer must pay special attention to nondeterministic asynchronous

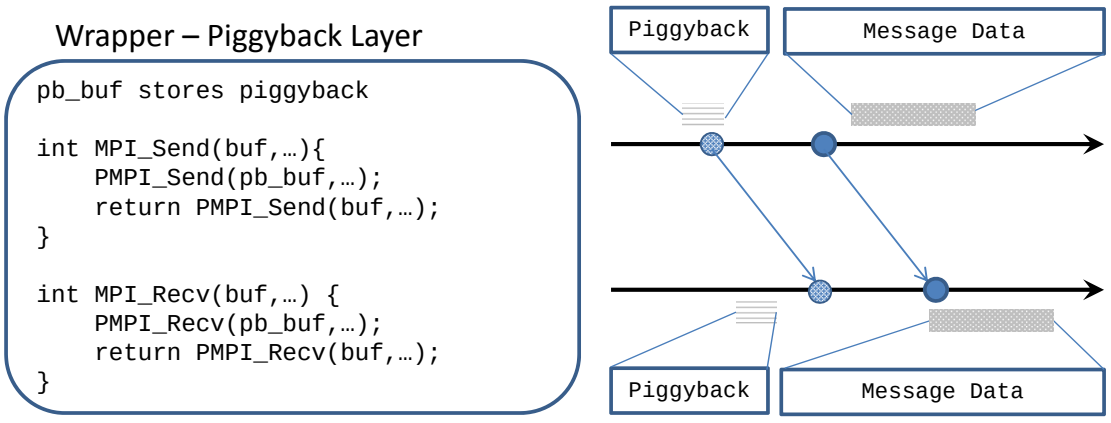

Figure 2.13: Separate message piggyback

wildcard receives since the sender of the message is not known at the time of issuing the receives. In such cases, the request handle for the piggyback is usually posted at the time of finishing the communication (e.g., `MPI_Wait`) [51]. However, we can show that the separate message piggybacking scheme as described does not correctly handle piggyback in the presence of wildcard receives. Consider the example shown in Figure 2.14 where the executed code shows all MPI calls being executed when the user code is linked together with a piggyback layer implementing the two message protocol. We first consider the case where the piggyback messages are transmitted over the same communicator with the original messages. The starred and italicized text indicates the extra messages that the piggyback layer inserts. Since they are from the same communicator, the piggyback message of the first `MPI_Isend`() ends up being received by the second `MPI_Irecv` of process P_0, which is erroneous.

We now consider the case in which the piggyback layer transmits the piggyback messages in a different communicator. This means for each communicator in the program, it would need to create a corresponding shadow communicator to send piggyback data. Consider the example shown in Figure 2.15, which is slightly different from the earlier example (the `MPI_Wait` calls are in different order with respect to the nonblocking sends). In this example, `pbcomm` is the piggyback communicator corresponding to `comm`. It is

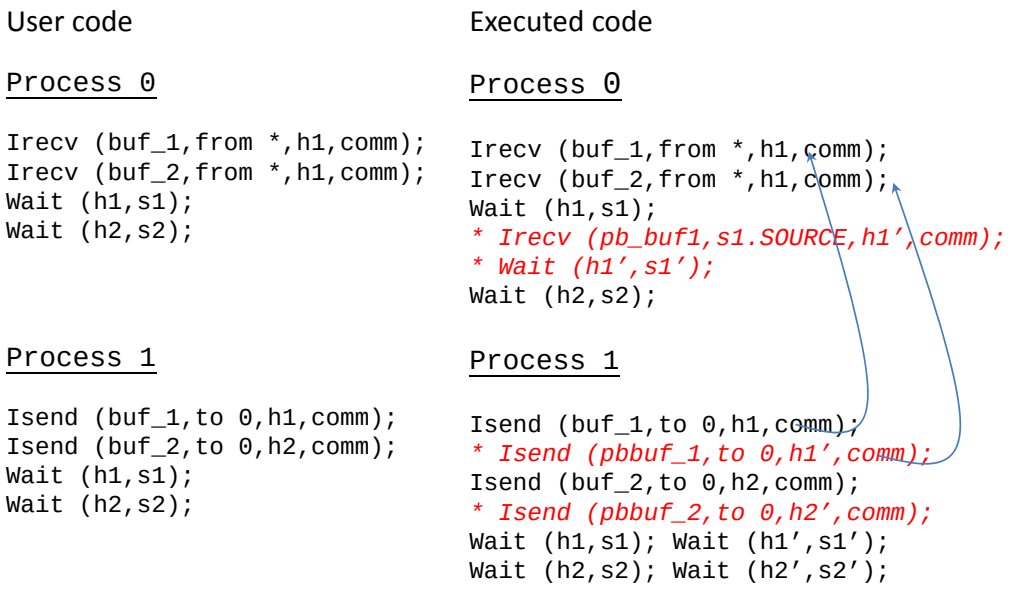

Figure 2.14: Separate message piggyback issue on the same communicator

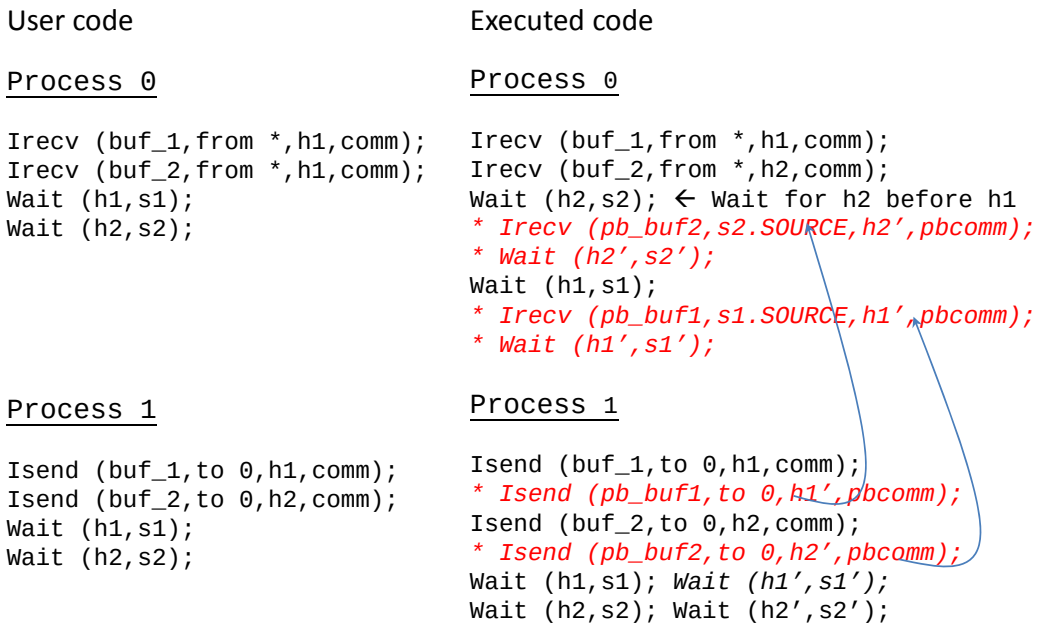

Figure 2.15: Separate message piggyback issue on different communicators

clear from the figure that the piggyback layer will end up associating the piggyback of the second message to the first message, which is also erroneous. This is due to the fact that one cannot post the piggyback receive requests immediately after the application receive requests because the sender of the message received by a nonblocking wildcard receive is not known until after the corresponding wait has completed.

Even with these shortcomings, separate message piggybacking remains a useful mechanism to attach and to receive piggyback data with collective operations. In fact, it is currently the only known method of transmitting piggyback data with collective operations (short of modifying the MPI library or breaking up the collective operations into point-to-point operations).

2.4.3 Datatype Piggyback

Another type of piggyback mechanism favored by many tools is datatype piggybacking [50, 54]. In this scheme, a new datatype is created by `MPI_Type_struct` for every send and receive operation. The new datatype combines a pointer to the main message buffer and a pointer to the current piggyback buffer. Figure 2.16 illustrates the mechanism of datatype piggybacking. In order to handle partial receives correctly, the piggyback data should be placed before the message data.

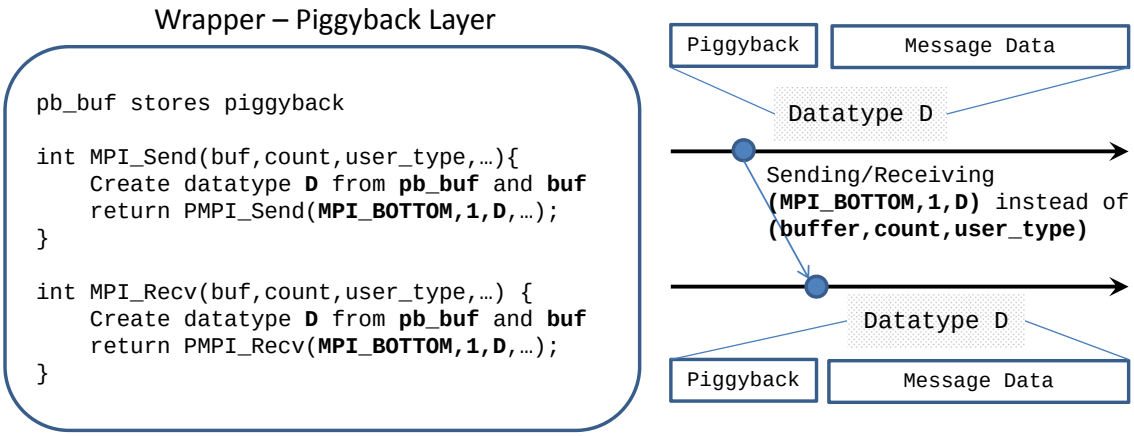

Figure 2.16: Datatype piggyback

Datatype piggybacking offers a compromise between buffer attachment piggybacking and separate message piggybacking. It does not suffer from the high bandwidth overhead and it correctly addresses piggybacking for nondeterministic receives. However, it does have several drawbacks: (i) moderate overhead due to excessive datatype creation (a new datatype has to be created for every send and receive operation), (ii) it is very difficult to implement piggyback for collective operations.

CHAPTER 3

MPI MATCHES-BEFORE

Using happens-before to track causality is an essential part of dynamic verification for parallel programs in general and MPI programs in particular. Unfortunately, the complex semantics of MPI allows many different types of interactions between events, many of which cannot be captured sufficiently by the traditional Lamport happens-before order that we discussed earlier. In this chapter we will discuss the issues of applying the happens-before order to MPI programs and introduce the MPI matches-before order which addresses the shortcomings.

3.1 Our Computational Model

A message passing program consists of sequential processes $P_0, P_1, \ldots, P_n$ communicating by exchanging messages through some communication channels. The channels are assumed to be reliable and to support the following operations:

- `send(dest,T)` - send a message with tag `T` to process `dest`. This operation has similar semantics to the `MPI_Send`, which means it has asynchronous behavior. That is, the call can complete before a matching receive has been posted.
- `ssend(dest,T)` - the synchronous version of `send`. This call only returns when the receiver has started to receive the message. In most MPI implementations, the receiver sends an *ack* to the sender to indicate that it has begun the receiving process.
- `recv(src,T)` - receive a message with tag `T` from process `src`. When `src` is `MPI_ANY_SOURCE` (denoted as $*$), any incoming message sent with tag `T` can be received (a *wildcard receive*).
- `isend(dest,T,`h`)` - the nonblocking version of `send`. The request handle h allows the call to be awaited for completion later. Similar to `send`, this call has an asynchronous behavior. The completion of the call (by a `wait`) only indicates that the buffer can be safely reused.

- `issend(dest,T,`h`)` - the synchronous version of `isend`. The completion of this call indicates that the receiver has started to receive the message.
- `irecv(src,T,`h`)` - the nonblocking version of `recv`. The request handle h allows the call to be awaited for completion later.
- `wait(`h`)` - wait for a nonblocking communication request until it completes. According to MPI semantics, *relevant piggyback information for a nonblocking receive cannot be accessed until the wait call.* Similarly, for a nondeterministic nonblocking receive, the source field (identity of the sending process) can only be retrieved at the wait.
- `barrier` - all processes have to invoke their barrier calls before any one process can proceed beyond the barrier.

For illustrative purposes, we abstract away the buffer associated with all send and receive events since it does not affect our algorithm. Further, we assume that all these events happen in the same communicator and that `MPI_ANY_TAG` is not used. We also do not consider collective operations other than `MPI_Barrier`. *Our implementation, however, does take into account all these possibilities.*

3.2 Issues Applying Happens-Before to MPI

We briefly go over how one might apply the traditional vector clocks algorithm to the example in Figure 3.1 to conclude that the first wildcard receive in P_0 can match either send from P_1 or P_2 and also why the Lamport clocks algorithm fails to do so.

Assuming the first receive from P_0 matches with the send from P_1 and the second receive from P_0 matches with the send from P_2, we want to know if the vector clocks algorithm can determine whether the first receive from P_0 could have received the message sent from P_2. Using the clock updating rules from the vector clocks algorithm described earlier, P_0's first receive's vector timestamp would be $[1, 1, 0]$ while the send from P_2 would have $[0, 2, 2]$. Clearly, the send and the receive are *concurrent* and thus, the send is a potential match to the receive.

In contrast, if we apply the Lamport clocks algorithm to this example, P_0's first receive event would have a clock of 1 while the send from P_2 would have a clock of 3. The algorithm could not determine whether the two events have any causal relationship. Hence, it cannot safely flag the send from P_2 as a potential match to the first receive from P_0. One can observe that the communication between P_1 and P_2 in this example has no

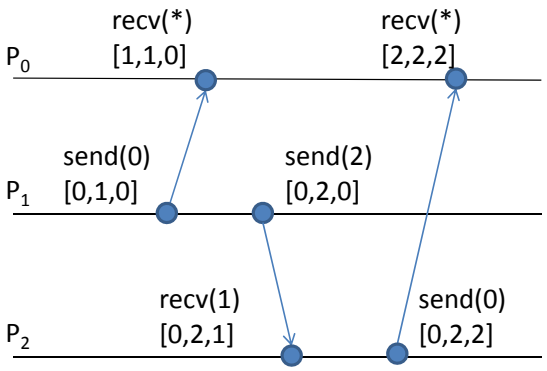

Figure 3.1: Wildcard receive with two matches

effect on P_0, yet the matching causes a clock increase which prevents the determination at P_0 of the causality between the first wildcard receive and P_2's send.

Now consider the example shown in Figure 3.2. Assuming the `irecv` by P_1, denoted as r, matches to the `isend` from P_0, we will apply the vector clocks algorithm to figure out whether the `isend` from P_2 , denoted as s, can be safely flagged as a potential match to r. By using the vector clock updating rules and considering `barrier` as a synchronization event where all processes synchronize their clocks to the global maximum, the clocks for r and s would be $[1,0,0]$, and $[1,0,1]$, respectively. This means $r \xrightarrow{\text{hb}_{\text{VC}}} s$ and thus the algorithm fails to recognize s as a potential match to r.

Clearly, the notion of *happening* and the corresponding *happens-before* order are insufficient for capturing all behaviors of MPI programs. We need a new model that allows us to completely capture the ordering of all events within an MPI program execution.

3.3 Matches-Before

We first consider the different possible states of an MPI operation *op* after a process invokes *op*:

P_0	P_1	P_2
`Isend(to` P_1`,22)`	`Irecv(from:*,x)`	`Barrier`
`Barrier`	`Barrier`	`Isend(to` P_1`,33)`
`Wait()`	`Recv(from:*,y)`	`Wait()`
	`if(x==33) ERROR`	
	`Wait()`	

Figure 3.2: Counterintuitive matching of nonblocking receive

- *issued* - *op* attains this state immediately after the process invokes it. All MPI calls are issued in program order.
- *matched* - We define this state in Definition 3.1.
- *returned* - *op* reaches this state when the process finishes executing the code of *op*.
- *completed* - *op* reaches this state when *op* no longer has any visible effects on the local program state. All blocking calls reach this state immediately after they return while nonblocking calls reach this state after their corresponding waits return.

Of these, only the *issued* and *matched* states have significant roles in our algorithms; nonetheless, we included all possible states for completeness. The *matched* state is central to our protocols and is described in further details below.

Definition 3.1 An event e in an MPI execution attains the *matched* state if it satisfies one of these conditions:

- e is an issued sending event of message m and the destination process has started to receive m through some event e'. e is said to have matched with e'. The receiving process is considered to have started the receive process when we can (semantically) determine from which of the send events it will receive the data. The timing of the completion of the receiving process is up to the MPI runtime and is not relevant to this discussion. e and e' in this case are considered to be in a *send-receive match-set*.
- e is a receive event that marks the start of reception. If e is a wildcard receive, we denote $e.src$ as the process with which e matched.
- e is a `wait(`h`)` call whose pending receive request associated with h has been matched. For an `isend`, the wait can attain the *matched* state upon completion while the `isend` still has not matched (i.e., it is buffered by the MPI runtime). A *matched* `wait` is the only element in its match-set (a *wait match-set*).
- e is a barrier and all processes have reached their associated barrier. e is said to have matched with e' if they are in the same set of barriers. All participating `barriers` are in the same match-set (a *barrier match-set*).

While it is straightforward to determine the matching point of synchronous calls `recv`, and `barrier`, the situation is more complex when it comes to nonblocking calls. The assumption that all nonblocking calls would attain the matched state exactly at their corresponding `wait` calls is incorrect. We explained the situation with the `isend` call earlier. Figure 3.3 shows another situation in which the first `irecv` call from process $P2$ can attain the matched state anywhere from its issuance to right before the `recv` call

returns (the arrow in the figure shows the interval during which the call can attain the matched state), which could be much earlier than the completion of its corresponding `wait`. This is due to the nonovertaking rule of the MPI standard.

Let E be the set of events produced in an execution, where each $e \in E$ is a *match* event as per Definition 3.1. We represent this execution as $P = \langle E, \xrightarrow{\text{mb}} \rangle$, where $\xrightarrow{\text{mb}}$ is the *matches-before* relation over E defined as follows. Consider two distinct events e_1 and e_2 in E; $e_1 \xrightarrow{\text{mb}} e_2$ if and only if one of the following conditions holds:

- $C1$. e_1 and e_2 are two events from the same process where e_1 is either a `ssend`, `recv`, `wait`, or `barrier`, and e_2 is issued after e_1.
- $C2$. e_1 is a nonblocking receive and e_2 is the corresponding wait.
- $C3$. e_1 and e_2 are send events from the same process i with the same tag, targeting the same process j and e_1 is issued before e_2. *This is the nonovertaking rule of MPI for sends.* The sends can be either blocking/nonblocking.
- $C4$. e_1 and e_2 are receive events from the same process i with the same tag, either e_1 is a wildcard receive or both are receiving from the same process j, and e_1 is issued before e_2. *This is the nonovertaking rule of MPI for receives.* The receives can be blocking or nonblocking.
- $C4'$. e_1 and e_2 are receive events from the same process i in which e_2 is a wildcard receive; e_2 is issued after e_1, $e_2.tag = e_1.tag$, and $e_2.src = e_1.src$. $C4'$ is a special case of $C4$ in which the $\xrightarrow{\text{mb}}$ relationship between e_1 and e_2 can only be determined after e_2 attains the matched state.
- $C5$. e_1 and e_2 are from two different processes and there are events e_3 and e_4 such that $e_1 \xrightarrow{\text{mb}} e_3$, $e_4 \xrightarrow{\text{mb}} e_2$, and furthermore e_3 and e_4 are in the same match-set, and e_3 is not a receive event (i.e., e_3 is either a `send`, `isend`, or `barrier`). Figure 3.4 illustrates this transitivity condition. The two shaded areas in the figure show a send-receive match-set and a barrier match-set while the dashed arrows show the matches-before relationship between events in the same processes. Condition $C5$

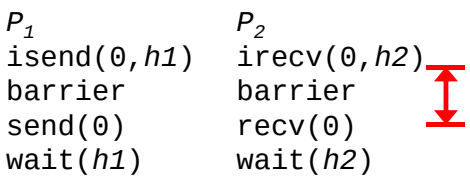

Figure 3.3: Nonovertaking matching of nonblocking calls

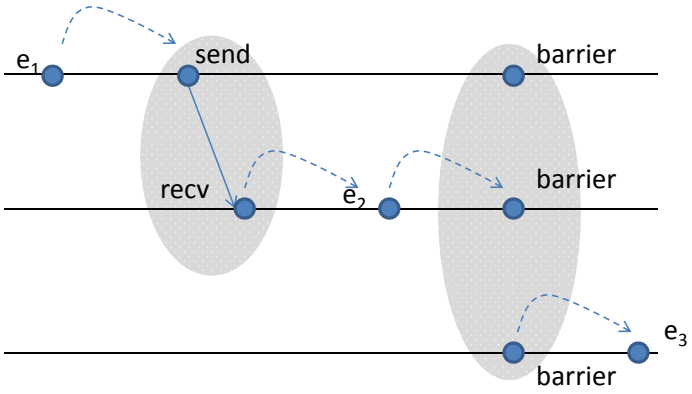

Figure 3.4: Transitivity of matches-before

allows us to infer that $e_1 \xrightarrow{\text{mb}} e_2$ and $e_2 \xrightarrow{\text{mb}} e_3$.

- $C6$. There exists an event e_3 such that $e_1 \xrightarrow{\text{mb}} e_3$ and $e_3 \xrightarrow{\text{mb}} e_2$ (transitive order). In Figure 3.4, condition $C5$ and $C6$ allow us to infer that $e_1 \xrightarrow{\text{mb}} e_3$.

Corollary 3.1 If e_1 and e_2 are two events in the same match-set, neither $e_1 \xrightarrow{\text{mb}} e_2$ nor $e_2 \xrightarrow{\text{mb}} e_1$ holds.

Corollary 3.2 If e_1 and e_2 are two events in the same process and e_1 is issued before e_2, then $e_2 \xrightarrow{\text{mb}} e_1$ is false.

In addition to the $\xrightarrow{\text{mb}}$ relationship for two events, we also define the $\xrightarrow{\text{mb}}$ relationship between X and Y where either X, Y, or both are match-sets. In which case, $X \xrightarrow{\text{mb}} Y$ if and only if one of the following conditions holds:

- $C7$. X is an event e_1, Y is a send-receive match-set, e_2 is the send event in Y, and $e_1 \xrightarrow{\text{mb}} e_2$.
- $C8$. X is an event e_1, Y is either a barrier match-set or a wait match-set, for all events e_2 in Y: $e_1 \xrightarrow{\text{mb}} e_2$.
- $C9$. X is a send-receive match-set, e_1 is the receive event in X, Y is an event e_2, and $e_1 \xrightarrow{\text{mb}} e_2$.
- $C10$. X is a send-receive match-set in which the send e_1 is a synchronous send, e_2 is the corresponding receive in the same match-set, Y is an event e_3, and $e_1 \xrightarrow{\text{mb}} e_3 \wedge e_2 \xrightarrow{\text{mb}} e_3$.
- $C11$. X is a barrier match-set or a wait match-set and Y is an event e_2, and there exists some event e_1 in X: $e_1 \xrightarrow{\text{mb}} e_2$.
- $C12$. X and Y are both match-sets, there is some event e_1 in X such that $e_1 \xrightarrow{\text{mb}} Y$.

Definition 3.2 Two events e_1 and e_2 are considered *concurrent* if they are not ordered by $\xrightarrow{\text{mb}}$. Let $e_1 \overset{mb}{\not\rightarrow} e_2$ denote the fact that e_1 is not required to match before e_2; then e_1 and e_2 are concurrent if and only if $e_1 \overset{mb}{\not\rightarrow} e_2 \wedge e_2 \overset{mb}{\not\rightarrow} e_1$.

3.4 Discussion

We have provided the notion of matches-before, which allows us to correctly capture the causality between events in an MPI execution. We have also defined the concept of *match-set*, which treats the matching action between a send and a receive as a single event itself. This is in contrast with most protocols based on the traditional Lamport clocks and vector clocks, which consider the sending event to happens-before the receive event. In the next chapter, we will introduce the **P**artial **O**rder Avoid **E**lusive Interleavings (POE) algorithm, which uses a centralized version of the matches-before order. This centralized version of the matches-before order allows for easy implementation but it does not scale well. We later introduce the lazy update algorithms that uses the matches-before order introduced in this chapter as the basis to provide scalable causality tracking for MPI.

CHAPTER 4

CENTRALIZED DYNAMIC VERIFICATION FOR MPI

Since we have adopted the matches-before relationship in place of the traditional Lamport happens-before, we also need new clock updating algorithms that correctly characterize the causality information between events in an MPI execution based on $\xrightarrow{\text{mb}}$. In this chapter we present the first approach, which uses a centralized scheduler to maintain a global view of all interactions between all MPI calls. This global view enables the scheduler to maintain the matches-before relationship in order to determine whether a nondeterministic event can have multiple different outcomes, and enforce those outcomes through replay.

This chapter only summarizes some of the key concepts of ISP. My work on ISP has mainly focused on improving ISP's scalability and usability, for which I provide the details in Section 4.2.

4.1 ISP

ISP, which stands for **I**n-**S**itu **P**artial order, is a dynamic verifier for MPI programs which is driven by the POE algorithm. ISP verifies MPI programs for deadlocks, resource leaks, type mismatches, and assertion violations. ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI runtime. This is accomplished by the two main components of ISP: the ISP Profiler and the ISP Scheduler. Figure 4.1 provides the overview of the ISP tool.

4.1.1 The ISP Profiler

The interception of MPI calls is accomplished by compiling the ISP Profiler together with the target programs source code. The profiler uses the MPI profiling interface (PMPI). It provides its own version of MPI_f for each corresponding MPI function f. Within each of these MPI_f, the profiler communicates with the scheduler using either

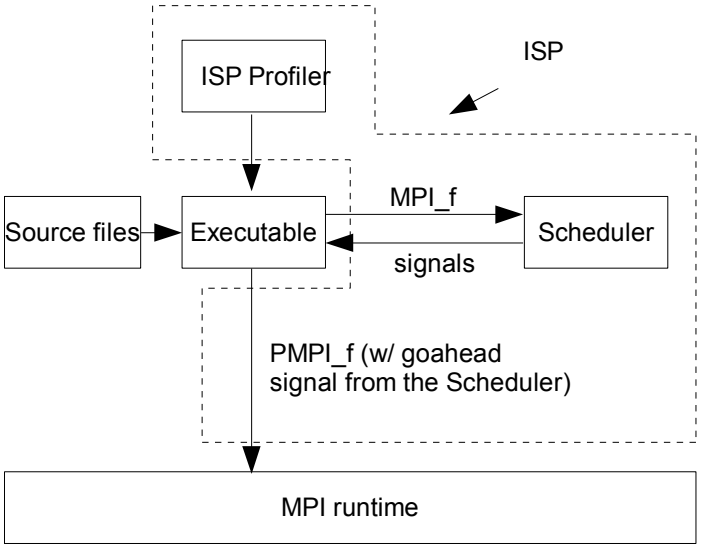

Figure 4.1: ISP framework

TCP sockets or Unix sockets to send information about the MPI call the process wants to make. The profiler will then wait for the scheduler to make a decision on whether to send the MPI call to the MPI library or to postpone it until later. When the permission to fire f is given from the scheduler, the corresponding PMPI_f will be issued to the MPI runtime. Since all MPI libraries come with functions such as PMPI_f for every MPI function f, this approach provides a portable and light-weight instrumentation mechanism for MPI programs being verified.

4.1.2 The ISP Scheduler

The ISP scheduler carries out the verification algorithms. Since every process starts executing with an `MPI_Init`, every process invokes the `MPI_Init` provided by the profiler. This initialization phase of the profiler involves establishing a TCP connection with the scheduler and communicating its process rank to the scheduler. The TCP connection is used for all further communication between the process and the scheduler. The scheduler maintains a mapping between the process rank and its corresponding TCP connection. Once the connection with the scheduler is established, the processes execute a `PMPI_Init` into the MPI library. The processes finally return from the `MPI_Init` of the profiler and continue executing the program. Whenever a process wishes to execute an MPI function,

it invokes the MPI_f of the profiler, which communicates this information to the scheduler over the TCP connection. The profiler does not always execute the PMPI_f call into the MPI library when it calls the profilers MPI_f. For nonblocking calls like `MPI_Isend` and `MPI_Irecv`, the profiler code sends the information to the scheduler and stores this information in a structure in the profiler and returns. When the process executes a fence instruction like `MPI_Wait`, the scheduler makes various matching decisions and sends a message to the process to execute the `PMPI_Isend` (or other nonblocking functions) corresponding to the Wait call. The MPI library is not aware of the existence of `MPI_Isend` until this time. Eventually, the scheduler sends a message to the process to execute the `PMPI_Wait`, at which time the process returns. It must be noted that the scheduler will allow a process to execute a fence MPI function only when the `Wait` can complete and hence return. Otherwise, the scheduler will detect a deadlock.

4.1.3 The POE Algorithm

The ISP scheduler implements the POE (**P**artial **O**rder avoiding **E**lusive interleavings) algorithm [65]. We first provide the intuition for the POE algorithm by considering the example in Figure 4.2, which is the same crooked barrier example in Chapter 3. The scheduler allows us to have the absolute control of the MPI runtime and gives us the ability to only execute the MPI calls at our discretion as long as the MPI semantics is preserved. In that case, instead of executing the matching between the `isend` of P_0 and the `irecv` of P_1, we delay the `irecv` call and execute other MPI calls first, until the process invoke some MPI calls f which requires that the `irecv` matches before it (e.g., the `recv` call in the example). Clearly, by delaying the `irecv` and executing the `barrier` first, we can now see both of the `isend`'s coming from P_0 and P_2 as possible matches for the `irecv` from P_1. We now briefly describe the POE algorithm. The formal description and the proof of correctness are available in [63].

The POE algorithm works as follows. There are two classes of statements to be executed: (i) those statements of the embedding programming language (C/C++/Fortran) that do not invoke MPI commands, and (ii) the MPI function calls. The embedding statements in an MPI process are local in the sense that they have no interactions with those of another process. Hence, under POE, they are executed in program order. When an MPI call f is encountered, the scheduler records it in its state; however, it does not (necessarily) issue this call into the MPI runtime. (Note: When we say that the scheduler

P_0	P_1	P_2
`Isend(to` P_1`,22)`	`Irecv(from:*,x)`	`Barrier`
`Barrier`	`Barrier`	`Isend(to` P_1`,33)`
`Wait()`	`Recv(from:*,y)`	`Wait()`
	`if(x==33) ERROR`	
	`Wait()`	

Figure 4.2: MPI example to illustrate POE

issues/executes MPI call f, we mean that the scheduler grants permission to the process to issue the corresponding PMPI_f call to the MPI runtime.) This process continues until the scheduler arrives at a *fence*, where a fence is defined as an MPI operation that cannot complete after any other MPI operation following it. The list of such fences includes all MPI blocking calls such as `MPI_Wait`, `MPI_Barrier`. When all processes reach their fences, the POE algorithm now forms *match-sets* as described earlier in Chapter 3. Each match-set is either a single big-step move (as in operational semantics) or a set of big-step moves. A *set* of big-step moves results from dynamically rewriting a wildcard receive. Each big-step move is a set of actions that are issued collectively into the MPI run-time by the POE-scheduler (we enclose them in $\langle\langle..\rangle\rangle$). In the example of Figure 4.2, these are all possible match-sets. Note that we rewrite the wildcard into each specific process according to the matching send.

• The set of big-step moves

{

$\langle\langle$ P_0`'s isend(to` P_1`),` P_1`'s irecv(from` P_0`)` $\rangle\rangle$,

$\langle\langle$ P_2`'s isend(to` P_1`),` P_1`'s irecv(from` P_2`)` $\rangle\rangle$,

}

• The single big-step move

$\langle\langle$ `Barrier,Barrier,Barrier` $\rangle\rangle$

The POE algorithm executes all big-step moves (match sets). The execution of a match-set consists of executing all of its constituent MPI operations (firing the PMPI versions of these operations into the MPI runtime). The *set of big-step moves* (set of match sets) is executed only when no ordinary big-step moves are left. In our example, the big-step move of barriers is executed first. This priority order guarantees that a representative sequence exists for each possible interleaving [65].

Once only a set of big-step moves are left, each member of this set (a big-step move) is fired. The POE algorithm then resumes from the resulting state.

In our example, each big-step moves in the set

{

⟨⟨ P_0's isend(to P_1), P_1's irecv(from P_0) ⟩⟩,

⟨⟨ P_2's isend(to P_1), P_1's irecv(from P_2) ⟩⟩,

}

is executed, and the POE algorithm is invoked after each such big-step move.

Thus, one can notice that the POE scheduler never actually issues into the MPI run-time any wildcard receive operations it encounters. It always dynamically rewrites these operations into receives with specific sources, and pursues each specific receive paired with the corresponding matching send as a match-set in a depth-first manner.

4.1.4 ISP Evaluation

We present an evaluation of ISP with Marmot [35], a popular MPI correctness checking tool. Marmot detects deadlocks using a timeout mechanism. Marmot also uses the MPI Profiling Interface to trap MPI calls. The timeout mechanism works by enforcing an interval that represents Marmot's estimate of the computation time between two successive MPI calls. When a process does not execute any MPI call after the timeout interval, Marmot signals a deadlock warning. For the experiment, we apply both ISP and Marmot on the Umpire test suite [68] and report the results on selected benchmarks in Table 4.1. The full set of experiments is also available [5].

Table 4.1 has three columns. The first column provides the Umpire benchmark programs. The second column shows the result of running the Umpire benchmark on

Table 4.1: Comparison of POE with Marmot

Umpire Benchmark	POE	Marmot
`any_src-can-deadlock7.c`	Deadlock Detected 2 interleavings	Deadlock Caught in 5/10 runs
`any_src-can-deadlock10.c`	Deadlock Detected 1 interleaving	Deadlock Caught in 7/10 runs
`basic-deadlock10.c`	Deadlock Detected 1 interleaving	Deadlock Caught in 10/10 runs
`basic-deadlock2.c`	No Deadlock Detected 2 interleavings	No Deadlock Caught in 20 runs
`collective-misorder.c`	Deadlock Detected 1 interleaving	Deadlock Caught in 10/10 runs

ISP executing the POE algorithm. We show the number of interleavings generated by POE. The last column shows the result of running the benchmark with Marmot. The benchmark is run multiple times on Marmot to evaluate the effectiveness of Marmot's deadlock detection scheme. Since Marmot's deadlock detection scheme relies on the deadlock's occurrence during a particular run, it cannot guarantee the detection of *possible deadlocks* due to nondeterminism. The basic-deadlock2.c example presents a deadlock scenario in which the deadlock only happens if the verification restricts the `MPI_Send` calls to having zero buffer space. Since POE is set to provide infinite buffering in this experiment, we do not report the deadlock here. Upon setting the available buffer space for `MPI_Send` to 0, the deadlock is caught.

4.2 ISP Scalability Issues

While the centralized scheduler easily maintains a complete global picture that facilitates the state space discovery process, it limits scalability. When the number of MPI calls becomes sufficiently large, the synchronous communication between the scheduler and the MPI processes becomes an obvious bottleneck. This section details our efforts in improving ISP's scalability and the lessons learned throughout the process.

4.2.1 The Scalability Challenge

We attempted to apply ISP on ParMETIS [12], which is a hypergraph partition library, to verify its routines for freedom of deadlocks as well as resource leaks. Verifying ParMETIS is a challenging task, not only because of its scale (AdaptiveRepart, one repartition routine provided by ParMETIS, has more than 12,000 lines of code between itself and its helper functions), but also because of the enormous number of MPI calls involved. In some of our tests, the number of MPI calls recorded by the ISP scheduler exceeds 1.3 million. This class of applications stresses both the memory usage overhead and the processing overhead of the scheduler.

Our attempt to improve ISP while working on the large code base of ParMETIS introduced several challenges at a pragmatic level. Since we did not have another MPI program debugger – and especially one that understands the semantics of our ISP scheduler that was itself being tweaked – we had to spend considerable effort employing low level debugging methods based on `printf`s and similar methods.

4.2.2 Memory Consumption

In order to replay the execution of the processes and correctly skip over all previous matching of sends/receives, ISP has to store all transitions (i.e., the MPI calls) for each process. This consumes a considerable amount of memory. The problem was not very apparent when we tested ISP with the Umpire test suite [68] and the Game of Life program [65], which made fewer than a hundred MPI calls in our testing. In our several first runs, ParMETIS exceeded all available memory allocations.

The problem was attributed to the storage taken by ISP's *Node* structure which maintains the list of transitions for each process. In addition, each transition maintained a list of ancestors which grew quadratically. We will describe our approach to handling this problem in Section 4.2.3.1.

Forming match sets is a central aspect of POE. One source of processing overheads in match set formation was located to be the compiler's inability to inline the `.end()` call in loops such as this:

```
for (iter = list.begin(); iter != list.end();
     iter++) {
    ... do something ...
}
```

Improvements at this level had marginal effects on ISP's performance.

4.2.3 Improvements to POE

4.2.3.1 Transitivity of Matches-Before

It became obvious that searching through hundreds of thousands of matches-before edges was having a huge effect on ISP's performance. We either needed to store less matches-before edges, or search through less matches-before edges. First, we exploit the fact that *ancestor* is a transitive binary relation, and store only the *immediate ancestor* relation. As the name suggests, immediate ancestor is the *transitive reduction* of the *ancestor* relation – i.e., the smallest binary relation whose transitive closure is *ancestor*. We then realized that the POE algorithm remained correct even if it employed immediate ancestors in match-set formation. The intuitive reason for this lies in the fact that whenever x is an ancestor of y and y is an ancestor of z, a match set involving y would be formed (and fired) before one involving z is formed (and fired).

The graph in Figure 4.3 shows the improvement of ISP in handling ParMETIS after switching over to the use of immediate ancestors. The testing setup we employed is similar to the `ptest` script provided in ParMETIS 3.1 distribution. To be more specific, our tests involve running rotor.graph, a file that represents a graph with about 100,000 nodes and 600,000 edges, through V3_RefineKWay, followed by a partitioning routine called V3_PartKway, then the graph is repartitioned again with AdaptiveRepart. The test completes by running through the same routine again with a different set of options. All tests were carried out on a dual Opteron 2.8 GHz (each itself is a dual-core), with 4 GB of memory. We also show in Table 4.2 the number of MPI calls this test setup makes (collectively by all processes) as the number of processes increases.

The comparison between the original ISP and the modified ISP (dubbed ISP v2 in this study) shows a huge improvement in ISP's processing time. In fact, without the use of immediate ancestors, ISP was not able to complete the test when running with eight processes. Even running one test for 4 processes already took well over a day! In contrast, ISP v2 finishes the test for 4 processes in 34 minutes.

With the change over from ancestors to immediate ancestors, we also made additional data structure simplifications, whose impact is summarized in the graph of Figure 4.4 (this version of ISP was termed ISP v3).

Even with these improvements, ISP was still taking considerable time to complete

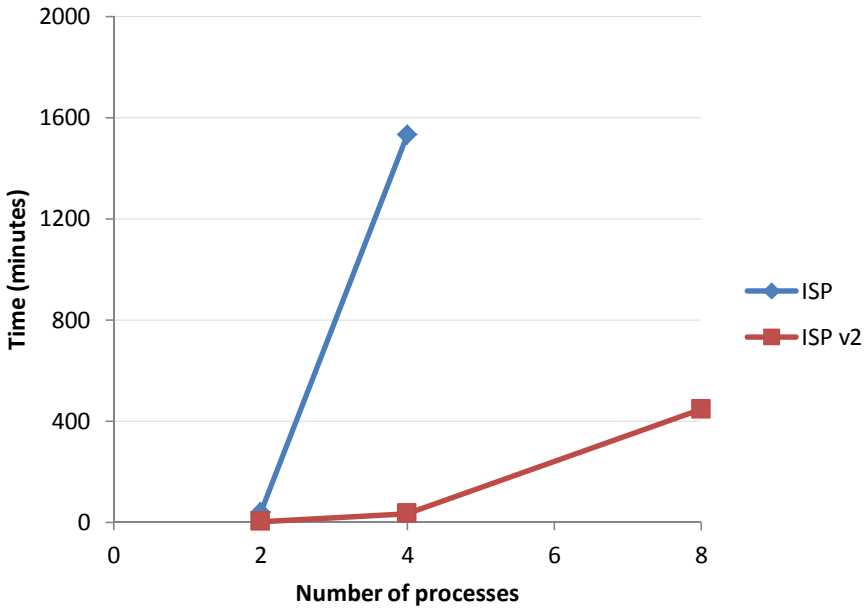

Figure 4.3: Improvements based on transitive reduction

Table 4.2: Number of MPI calls in ParMETIS

num. of procs	Total MPI calls
2	15,789
4	56,618
8	171,912
16	544,114
32	1,390,260

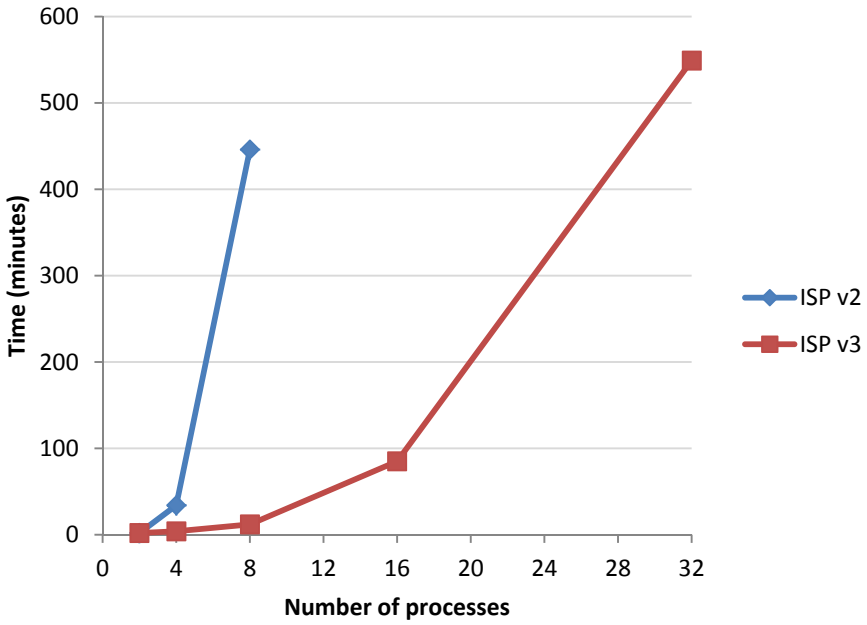

Figure 4.4: Improvements made by data structures changes

model checking ParMETIS for 32 processes, which is almost ten hours. This led us to consider parallelizing the search for ancestors.

4.2.3.2 Parallel-ISP

The discovery of where ISP spends most of its processing time leads us to the idea of parallelizing ISP's search for ancestors while building the match-sets. Recall that the MPI calls made by each process of the target program are represented by transition lists. The formation of match sets requires searching through *all* transition lists. Fortunately, these searches are independent of each other, and can be easily parallelized. There are several ways to parallelize this process: (i) make a distributed ISP where each ISP process performs the search for each transition list, or (ii) create a multithreaded-ISP where each thread performs the search, or (iii) use OpenMP to parallelize the search and let the

OpenMP run-time handle the thread creation. We opted for the OpenMP approach due to the fact that the POE scheduler is implemented with many `for` loops – a good candidate for parallelization using OpenMP.

We present the performance results of Parallel ISP vs. ISP v3 in Figure 4.5. Parallelization does not help ISP much when running with a small number of processes. However, when we verify up to 16 and 32 processes, the benefits of parallelization becomes more obvious (On average, Parallel-ISP was about 3 times faster than the serial ISP).

4.2.4 Discussion

Although ISP has been improved greatly to handle practical MPI programs. We still notice that as the number of processes increases, the performance of ISP degrades exponentially. We investigate the system load of ISP verifying ParMETIS with 32 processes and notice that the ISP Scheduler is taking almost all of the CPU time while the MPI processes are just waiting for the responses from the scheduler. This shows that ISP fails to exploit the distributed processing of all processes, which means it will become infeasible to verify large MPI applications beyond a few dozen processes. An early experimental version of ISP was developed in which MPI processes would be launched on different hosts and communicate with the scheduler through TCP sockets. The distributed launching mechanism effectively removes the resource constraints faced by

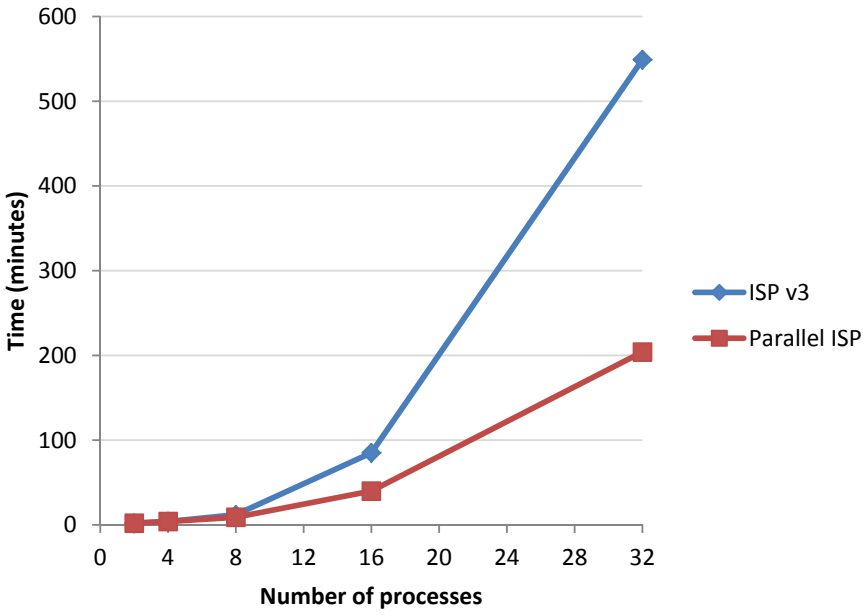

Figure 4.5: Improvements made by parallelization

launching all MPI processes within one single machine. However, we have run experiments that demonstrates that the main bottleneck lies in the scheduler and running the MPI processes in a distributed environment does little in speeding up the verification. In the next chapter we will present our distributed approach to address ISP's scalability issues.

CHAPTER 5

DISTRIBUTED DYNAMIC VERIFICATION FOR MPI

We have shown in the previous chapter that the centralized approach does not scale well beyond a few dozen processes. In order to maintain good scalability, the verification needs to exploit the processing power of all processes. In essence, a good algorithm has to run the verification in distributed fashion and cannot rely on a centralized scheduler. To this end, we propose two algorithms: the Lazy Lamport Clocks Protocol (LLCP) and the Lazy Vector Clocks Protocol (LVCP). These are the design goals of the protocols.

- *Scalable* - Many MPI applications today require at least some scale in order to run certain inputs due to memory size and other limits. Further, many bugs, including nondeterminism related bugs, are only manifest when a problem is run at large scale. Any protocol aiming at handling large scale MPI programs must be scalable as well. LLCP is very scalable compared to LVCP as demonstrated by our experimental results.
- *Sound* - We define a *sound* protocol to be one that does not force the match of events that cannot match. Clearly, this is a crucial goal; an unsound protocol can *cause* a deadlock in an otherwise deadlock-free MPI program! We argue that both LLCP and LVCP are sound.
- *Complete* - While it is challenging to design a causality tracking protocol that is both complete and scalable, we still want to have a protocol that is scalable and maintains completeness *in the most common usages.* In all our testing with real MPI programs, LLCP proved to be complete, that is *we did not discover any extra matches when we ran the same program under LVCP on the same test harness.* If completeness in all cases is a requirement, then LVCP should be used.

5.1 Lazy Lamport Clocks

5.1.1 Algorithm Overview

LLCP maintains the matches-before relationship between events by maintaining a clock in each process and associates each event with a clock value in a way that can help us order these events according to when they attain the matched state. Since the matches-before relationship describes the ordering for events inside a process and across processes, the algorithm needs to be able to offer such coverage. More specifically, given a wildcard receive r from process P_i and a send s targeting P_i which did not match with r, the protocol should allow us to figure out whether r and s have any matches-before relationship between them. If, for example, the successful completion of r triggers the issuance of s, then it is never the case that s could have matched with r. The intuitive way to do this is to have the protocol maintain the clock such that *if r triggers the issuance of some event e, then it must be the case that the clock of r is smaller than the clock of e*. Basically this means all outgoing messages after r from P_i need to carry some clock value (as piggyback data) higher than r.

The challenge of the protocol lies in the handling of nonblocking wildcard receives. As explained earlier in the example in Figure 3.2, a nonblocking wildcard receive from a process P_i could potentially be pending (not yet reach the matched state) until its corresponding wait is posted. However, we have also shown in Figure 3.3 that such a receive could also attain the matched state due to the nonovertaking semantics (which could be earlier than the posting of the wait). The protocol needs to know precisely the status of the receive to avoid sending the wrong piggyback data, which could lead to incorrect matching decisions.

5.1.2 Clock Update Rules

We now describe the protocol in detail through a set of clock updating rules. For simplicity, we assume the programs do not contain synchronous sends and discuss the handling of synchronous sends in Section 5.2.1.

- $R1$. Each process P_i keeps a clock LC_i, initialized to 0.
- $R2$. When a nonblocking wildcard receive event e occurs, assign LC_i to $e.LC$ and add e to the set of pending receives: $Pending \leftarrow Pending \cup \{e\}$.
- $R3$. When P_i sends a message m to P_j, it attaches LC_i (as piggyback data) to m (denoted $m.LC$).

- $R4$. When P_i completes a receive event r (either forced by a blocking receive or at the wait of a nonblocking receive as in Figure 3.3), it first constructs the ordered set $CompleteNow$ as follows: $CompleteNow = \{e \mid e \in Pending \wedge e \xrightarrow{\text{mb}} r\}$. The set $CompleteNow$ is ordered by the event's clock, where $CompleteNow[i]$ denotes the i^{th} item of the set and $\left|CompleteNow\right|$ denotes the total items in the set. Intuitively, this is the set of pending nonblocking receives that have matched before r due to the MPI nonovertaking rule. Since they have all reached the matched status, we need to update their clocks as well. Note that the ordering of the events in $CompleteNow$ is very important since all receives in $CompleteNow$ are also $\xrightarrow{\text{mb}}$ ordered by the nonovertaking semantics themselves. We can update the clocks using the following loop:

  for $i = 1$ TO $\left|CompleteNow\right|$ **do**
  $CompleteNow[i].LC = LC_i$
  $LC_i \leftarrow LC_i + 1$
  end for
  $Pending \leftarrow Pending \setminus CompleteNow$

  After this, the process associates the current clock with r: $r.LC \leftarrow LC_i$ and advances its clock to reflect the completion of a wildcard receive: $LC_i \leftarrow LC_i + 1$. Note that the clock assignment and advancement do not happen to those nonblocking receives that have their clocks increased earlier due to the *for* loop above. We can check this by detecting whether the current nonblocking receive is still in the $Pending$ set or not. Finally, the process compares its current clock with the piggybacked data from the received message and updates LC_i to $m.LC$ if the current clock is less than $m.LC$.
- $R5$. At `barrier` events, all clocks are synchronized to the global maximum of the individual clocks.

5.1.3 Match Detection

Rules $R2$ and $R4$ form the *lazy* basis of the protocol in the sense that a nonblocking wildcard receive r gets a temporary clock when it initially occurs in the process and gets its final clock when it finishes (either by its corresponding wait or by another receive r' for which $r \xrightarrow{\text{mb}} r'$).

Lemma 5.1 If $e_1 \xrightarrow{\text{mb}} e_2$ then $e_1.LC \leq e_2.LC$

Proof. We first consider the case when e_1 and e_2 are from the same process. Based on our definition of matches-before, event e_2 will always occur after event e_1. Since our algorithm never decreases the clock, it follows that $e_1.LC \leq e_2.LC$.

Now assume e_1 and e_2 are events from two different processes. Based on the definition of matches-before, there exist events e_3 and e_4 such that $e_1 \xrightarrow{\text{mb}} e_3$, $e_4 \xrightarrow{\text{mb}} e_2$, e_3 and e_4 are in a match-set, and e_3 is either an `isend`, `send`, or `barrier`. We recursively apply this process to (e_1, e_3) and (e_4, e_2) and construct the set $S = s_1, s_2, .., s_n$ in which $s_1 = e_1$, $s_n = e_2$, and other elements are either events or match-sets that satisfy $s_i \xrightarrow{\text{mb}} s_{i+1}$. In addition, s has to satisfy the following rule: for any pair of adjacent elements (s_i, s_{i+1}), there does not exist any event e such that $s_i \xrightarrow{\text{mb}} e$ and $e \xrightarrow{\text{mb}} s_{i+1}$. Note that the construction of S is possible based on our definition of $\xrightarrow{\text{mb}}$. Intuitively, S represents all hops between e_1 and e_2 if one is to follow the $\xrightarrow{\text{mb}}$ chain event by event.

Now consider any pair of events (s_i, s_{i+1}). They must be both events from the same process, in which case $s_i.LC \leq s_{i+1}.LC$, or either one has to be a match-set, or both are match-sets, in which case our piggyback ensures that $s_i.LC \leq s_{i+1}.LC$. Hence, the set S has the property that $s_1.LC \leq s_2.LC \leq .. \leq s_n.LC$, which means $e_1.LC \leq e_2.LC$. ■

Lemma 5.2 Assuming r is either a blocking receive or a nonblocking receive that is not pending, if $r \xrightarrow{\text{mb}} e$ then $r.LC < e.LC$.

Proof. If e is an event in the same process with r then rule $R2$ and $R4$ ensure that $r.LC < e.LC$. If e is not in the same process with r then based on the definition of $\xrightarrow{\text{mb}}$, there must be an event f from the same process with r such that $r \xrightarrow{\text{mb}} f \wedge f \xrightarrow{\text{mb}} e$, which means $r.LC < f.LC$ and by Lemma 5.1, $f.LC \leq e.LC$. Thus, $r.LC < e.LC$. ■

We now introduce the concept of *late* messages, which is essential for the protocol to determine if an incoming send can match an earlier wildcard receive. One can think of late messages as in-flight messages in the sense that these messages have already been issued at the point when a receive reaches the matched state. Consider the MPI program shown in Figure 5.1. The first wildcard receive of P_1 matched with the `send` from P_0 while the second wildcard receive matches the `send` from P_2. The clock value associated with each event according to our protocol is displayed in the square bracket. The shaded area represents the set of all events that are triggered by r (i.e., for all events e in that shaded area, $r \xrightarrow{\text{mb}} e$). The message from P_2 was not triggered by the matching of the first wildcard receive in P_1, despite being received within the shaded area. We call the message from P_2 a late message. At the reception of late messages, the protocol checks

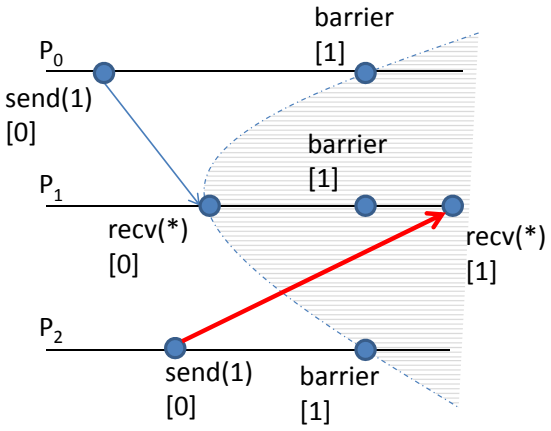

Figure 5.1: Late messages

whether they can be potential matches for receives that have matched earlier (in this figure, the late message from P_2 is a potential match).

Definition 5.1 A message m originating from process $m.src$ with timestamp $m.LC$ is considered *late* with respect to a wildcard receive event r (which earlier did not match with $m.src$) *iff* $m.LC \leq r.LC$. If message m is late with respect to r, it is denoted as $late(m, r)$.

We are now ready to devise the match detection rule:

Theorem 5.3 An incoming send s carrying message m with tag τ that is received by event r' in process P_i is a potential match to a wildcard receive event r with tag τ issued before r' if $(m.LC < r'.LC \wedge late(m, r))$

Proof. In order to prove that s is a potential match of r, we prove that s and r are concurrent, which means: $r \overset{mb}{\not\longrightarrow} s \wedge s \overset{mb}{\not\longrightarrow} r$. First we notice that $r \overset{\text{mb}}{\longrightarrow} r'$, which means r cannot be a pending event (due to rule $R4$). In addition, we also have $r.LC \geq s.LC$ since s is a late message. Using the contrapositive of Lemma 5.2, we infer that $r \overset{mb}{\not\longrightarrow} s$.

It is also the case that $s \overset{\text{mb}}{\not\longrightarrow} r$ because if $s \overset{\text{mb}}{\longrightarrow} r$, it must be the case that $s \overset{\text{mb}}{\longrightarrow} r'$ due to the transitive order rule of matches-before. This violates Corollary 3.1 which says that two events in the same match-set are not ordered by $\overset{\text{mb}}{\longrightarrow}$. ■

Let us now revisit the crooked barrier example introduced earlier in Figure 3.2 and show how the protocol applies (Figure 5.2 shows the same example with the clock values at the end of the execution). Using the LLCP clock update rules, the clock for the `irecv` by P_1 has a clock of 0 when it is issued and P_1 adds this `recv` to *Pending*. The `barrier` calls synchronize all clocks to the global maximum, which is 0. At the `recv` call, P_1

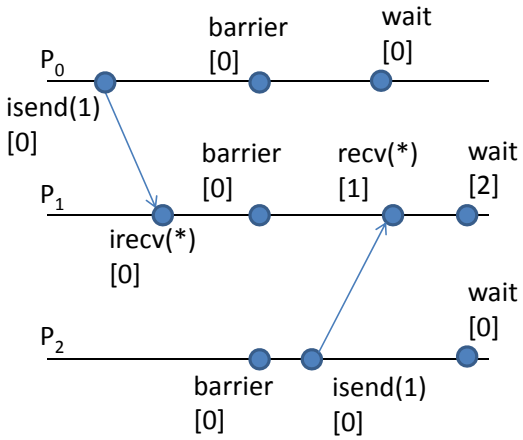

Figure 5.2: An example illustrating LLCP

applies rule $R4$ and constructs the *CompleteNow* set which consists of the `irecv`. Upon the completion of this step, the `irecv` has a clock of 0 and the `recv` has a clock of 1. Assuming that the `isend` from P_0 matches with the `irecv`, the `recv` call will match with the `isend` from P_2. The message from P_2 carries piggyback data of 0 and is flagged as a late message with respect to the `irecv` and is detected as a potential match (per Theorem 5.3).

It is important to note that the theorem only applies one way. That is, there might be potential matches that the LLCP misses. Consider the example in Figure 5.3 where it is easy to see by manual inspection that the send from P_0 is a potential match of the wildcard receive from P_2 (assuming the P_2's `recv` matches with P_1's `send`). However, the LLCP would fail to detect such a match since at the time of receiving P_0's `send`, the clock of P_0's send is 1, which is the same as the clock of P_2's `recv(0)`, and it does not satisfy the condition of Theorem 5.3.

This issue again reflects the disadvantage of Lamport clocks when there are multiple concurrent sources of nondeterminism. In general, omissions might happen when there are multiple sources of nondeterminism and processes for which the clocks are out of synchronization communicating with each other. Fortunately, this situation rarely happens in practice because most MPI programs have well-established communication patterns that do not have cross communications between groups of processes that generate relevant events before clock synchronization takes place. We will present our extension of LLCP to vector clocks that will address MPI programs with subtle communication patterns for which LLCP might not recognize all matches.

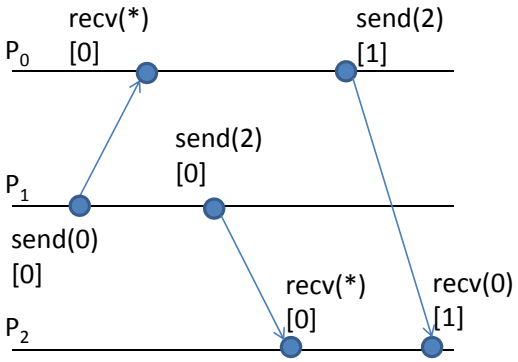

Figure 5.3: Omission scenario with LLCP

5.2 Lazy Vector Clocks

LLCP can be extended to use vector clocks. In the case of vector clocks, the rules remain similar while taking into account the fact that we are working with a vector of clocks (e.g., instead of incrementing a single clock, P_i now increments $VC_i[i]$). We shall now prove the updated lemmas and theorems that are based on LVCP.

Lemma 5.4 Assuming r is either a blocking receive or a nonblocking receive that is not pending in P_i: $r \xrightarrow{\text{mb}} e \Leftrightarrow r.VC[i] < e.VC[i]$.

Proof. The proof for $r \xrightarrow{\text{mb}} e \Rightarrow r.VC[i] < e.VC[i]$ is similar to the LLCP case and is omitted. We will now prove the converse.

Observe that in the LVCP, the only process that might increment $VC[i]$ is P_i (this is the fundamental difference between Lamport clock and vector clocks - which is also why the converse of this lemma does not hold for Lamport clocks). Thus, e is either an event that occurs in P_i after r completes (which is the point where $VC_i[i]$ becomes greater than $r.VC[i]$) or an event in another process that receives the piggybacked $VC_i[i]$ from P_i (either directly or indirectly via another process).

If e is an event that occurs in P_i after r completes and r is a blocking receive, we clearly have $r \xrightarrow{\text{mb}} e$ due to the definition of $\xrightarrow{\text{mb}}$. On the other hand, if r is a nonblocking receive, P_i will increase its clock (i.e., $VC[i]$) only in one of these scenarios:

- The corresponding wait for r is posted and r is still pending before the wait call.
- A blocking receive r' which satisfies $r \xrightarrow{\text{mb}} r'$ is posted and r is still pending before r'.
- A wait for a nonblocking receive r' which satisfies $r \xrightarrow{\text{mb}} r'$ is posted and r is still pending before r'.

Notice that in all of these scenarios, we need a blocking operation b such that $r \xrightarrow{\text{mb}} b$ to increase the clock of P_i. If $e.VC[i] \geq r.VC[i]$, it must be the case that e occurs after b. Hence, $b \xrightarrow{\text{mb}} e$ and by the transitive order rule, $r \xrightarrow{\text{mb}} e$. ■

Using the updated definition of *late* message (Definition 5.1) where $m.LC \leq r.LC$ is replaced by $m.VC[i] \leq r.VC[i]$, we now prove the LVCP matching theorem, which is stated as follows:

Theorem 5.5 An incoming send s carrying message m with tag τ being received by event r' in process P_i is a potential match to a wildcard receive event r with tag τ issued before r' if and only if $(m.VC[i] < r'.VC[i] \wedge late(m, r))$. In other words, all potential matches are recognized and there are no omissions.

Proof. The proof for the *if* part is similar to the LLCP proof and is omitted. We now prove the converse, which can be alternatively stated as: if $r \overset{mb}{\not\rightarrow} s \wedge s \overset{mb}{\not\rightarrow} r$ then $m.VC[i] < r'VC[i] \wedge late(m, r)$.

First we notice that due to the nonovertaking rule, $r \xrightarrow{\text{mb}} r'$, which gives us $r.VC[i] < r'.VC[i]$ according to Lemma 5.4. Now applying Lemma 5.4 to $r \overset{mb}{\not\rightarrow} s$, we obtain $m.VC[i] \leq r.VC[i]$, which means $m.VC[i] < r'.VC[i] \wedge late(m, r)$ (note that $m.VC[i]$ is the same as $s.VC[i]$ since $m.VC[i]$ is the piggybacked value attached to the message). ■

5.2.1 Handling Synchronous Sends

We briefly describe our approach to handle synchronous sends in MPI. Recall that a synchronous send s returns only when the corresponding receive call r has started to receive the message. Essentially, this means that if for any MPI events e, f such that $s \xrightarrow{\text{mb}} e$ and $r \xrightarrow{\text{mb}} f$, then $s \xrightarrow{\text{mb}} f$ and $r \xrightarrow{\text{mb}} e$.

5.2.1.1 Piggyback Requirements

Sending and receiving piggyback data for synchronous call is challenging and dependent on the MPI implementation. We have so far experimented with MPICH2 in which the sender of the synchronous call sends a Request-To-Send (RTS) packet to the receiver and waits for a Clear-To-Send (CTS) packet from the receiver indicating that the receiver is ready for the receiving process. We add a special field in the CTS packet to store the piggyback data and extract it on the sender side. Other MPI implementations use some similar rendezvous protocols and the same modifications can be applied. While such modifications can potentially limit portability, our experiments show that few MPI

applications use synchronous sends and for those that do, their communication patterns do not require this scheme of piggyback.

5.2.1.2 Algorithm Extension

We discuss the algorithm extension for the case of LLCP and omit the case of LVCP due to similarity. Consider the case where a synchronous send s matches with a receive r; the following extensions are made to the clock updating rules described in Section 5.1.2:

- Before s returns, it extracts the piggybacked clock c coming from the receiver's side from the CTS packet. If the receive that matches with s is a wildcard receive, it increments c by 1. Finally, it updates its clock to c if c is greater than its current clock. Intuitively, this rule ensures that any event e such that $s \xrightarrow{\text{mb}} e$ will have a higher clock than r.
- When the receiver side starts receiving the message from a synchronous send (by sending a CTS packet), we take no action if the receive is blocking receive; otherwise, if it is a nonblocking wildcard receive we add it into the set $MatchedWithSsend$. If a pending receive is in this set, any incoming message with a higher clock than the receive's clock at the time it is issued will not be counted as a potential match. This rule allows other eligible sends that are concurrent with the receive to be considered as potential matches.

The example in Figure 5.4 illustrates the extensions to handle synchronous sends. The extensions would allow LLCP to identify the `send` from P_0 correctly as a potential match and dismiss the `send` from P_3 as a potential match.

5.3 DAMPI: Distributed Analyzer for MPI

DAMPI (Distributed Analyzer for MPI) is the first dynamic MPI verifier that offers meaningful scalability: users can verify MPI codes within the parallel environment in which they develop and optimize them. In order to provide coverage over the nondeterminism space, DAMPI implements both LLCP and LVCP as its core modules and allows the users to use either protocol, depending on their needs. Many other optional error checking modules such as deadlock detection or resource leak detection are also available. In addition, DAMPI offers several search bounding heuristics that allow the user to focus the verification to regions of interests. We also report experimental results that demonstrate that DAMPI provides scalable, user-configurable coverage.

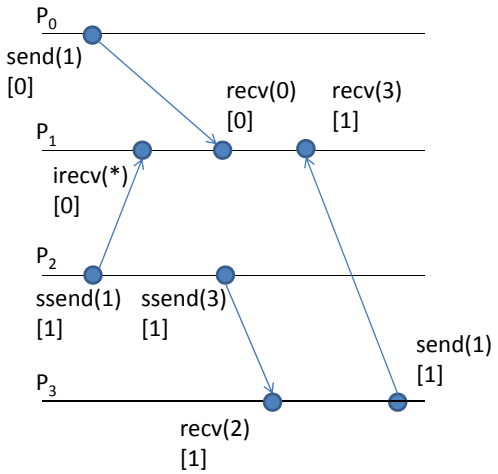

Figure 5.4: Handling synchronous sends

5.3.1 DAMPI Framework Overview

DAMPI has two main components: the DAMPI library, which is linked with the program to provide the MPI executable, and the scheduler, which provides the non-determinism coverage by restarting the processes to explore all possible interleavings. Figure 5.5 describes the overall framework of DAMPI.

5.3.1.1 The DAMPI Library

The DAMPI library is essentially a collection of several P^{N}MPI [53] modules providing core functionalities such as $\xrightarrow{\text{mb}}$ tracking or piggyback and several other optional error

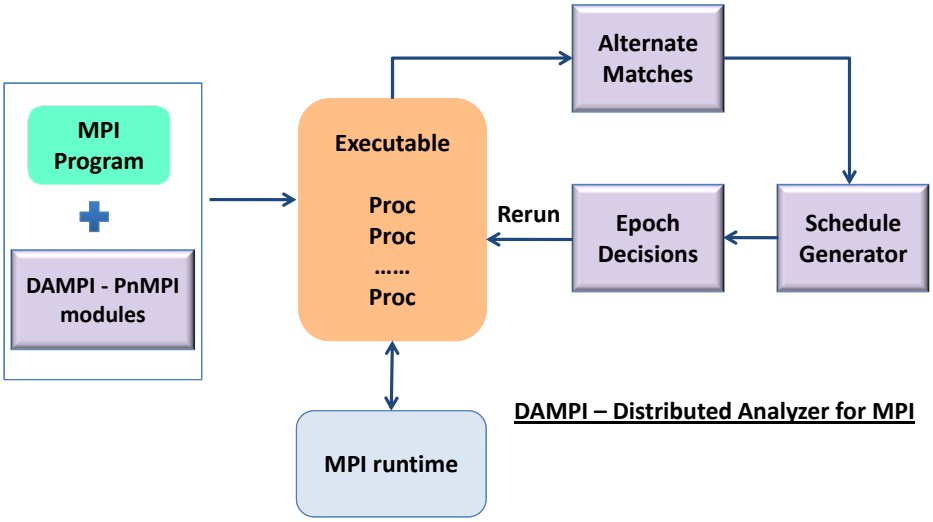

Figure 5.5: DAMPI framework

checking modules such as deadlock detection or resource leak tracking. Figure 5.6 provides an overview of the DAMPI library. The decision to implement DAMPI services as P^{N}MPI modules offers several advantages:

- Switching between different core services does not require recompilation. For example, to switch from using LLCP to using LVCP only requires the modification of the P^{N}MPI configuration file
- Turning on and off error checking modules also does not require recompilation. This is especially helpful during debugging sessions in which the users want to focus on several types of errors.
- Integration with other P^{N}MPI modules is possible and does not require the module developers to understand the detail of DAMPI.

5.3.1.2 The Scheduler

The scheduler operates in a postmortem manner and is responsible for replaying MPI programs according to the information collected by the MPI processes. In an MPI program where there are multiple possible outcomes due to nondeterminism, each process collects the information pertaining to those outcomes and outputs it to a database. The scheduler retrieves the information from the database and replays the program through the MPI runtime. During the replay process, the scheduler keeps collecting and processing information output by the processes to determine if a particular replay might lead to

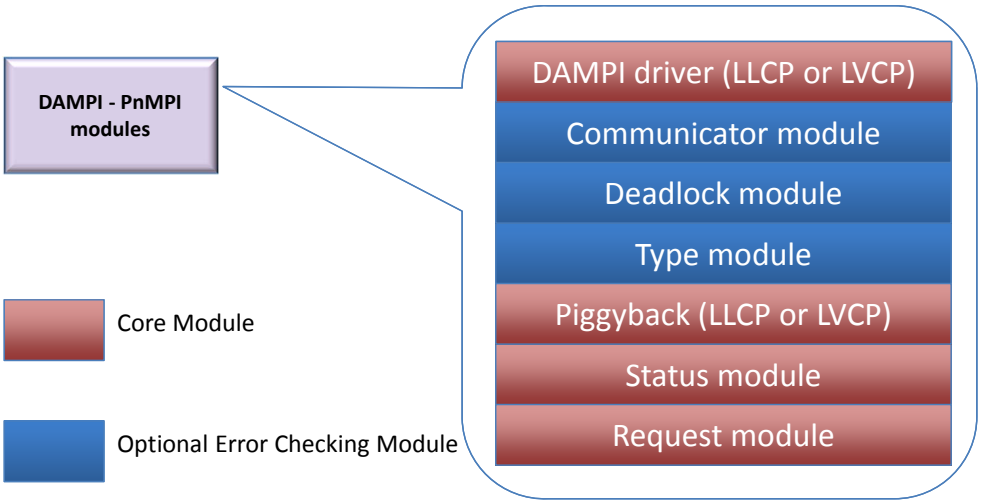

Figure 5.6: DAMPI library overview

additional executions.

5.3.2 Implementation Detail

5.3.2.1 Piggyback

The piggyback module implements piggybacking using a mixed scheme of both separate message piggybacking and datatype piggybacking. In particular, point-to-point messages rely on datatype piggybacking while collective messages use two message piggybacking. This mixed scheme of piggybacking allows us to achieve high performance without increasing code complexity (recall our discussion earlier in Chapter 2 that other than two message piggybacking, all other methods do not offer a simple mechanism to send piggyback data with collective calls). We use a single 32-bit integer per process to store the Lamport clock for LLCP. For LVCP, we use a vector of 32-bit integers for which the size of the vector equals the number of processes in the execution. These clock values are piggybacked on every outgoing message and are accessible on the receiving side after the calls attain the *complete* state.

Figure 5.7 provides the pseudocode for collectives under LLCP. Although different collective calls might have slightly different piggybacking schemes we only display the pseudocode for piggybacking inside an `MPI_Barrier` call and an `MPI_Bcast` call for simplicity. *Our implementation, however, does classify each collective properly based on their behaviors and executes the appropriate piggybacking scheme.*

MPI_Barrier(In:comm)

$in\_buf : int[piggyback\_size],\ out\_buf : int[piggyback\_size]$
$out\_buf \leftarrow piggyback\_buffer$
`PMPI_Allreduce`$(out\_buf, in\_buf, piggyback\_size$,`MPI_INT`,`MPI_MAX`$, comm)$
`PMPI_Barrier(comm)`
$piggyback\_buffer \leftarrow in\_buf$

MPI_Bcast(In:buf,count,dtype,root,comm)

$temp\_buf \leftarrow piggyback\_buffer$
`PMPI_Bcast`$(temp\_buf, piggyback\_size$,`MPI_INT`$, root, comm)$
if $myrank \neq root$ **then**
  $piggyback\_buffer \leftarrow max(piggyback\_buffer, temp\_buf)$
end if
`PMPI_Bcast`$(buf, count, dtype, root, comm)$

Figure 5.7: Packing and unpacking piggyback data - collective

Figure 5.8 provides the pseudocode for packing and unpacking piggyback data using the datatype piggybacking mechanism in which both the sending and the receiving side invoke the same procedure for packing and unpacking the piggyback data. On the sender side, the piggyback data are copied to a temporary buffer whose address is passed to the datatype construction. On the receiver side, the piggyback data are copied over to the temporary buffer during the receiving process and later is appended to the `status` field so that it can be accessed by other modules.

Figure 5.9 and 5.10 provide the pseudocode for handling the piggyback data in `MPI_Send` and `MPI_Isend`, respectively.

Figure 5.11, 5.12, and 5.13, illustrate the handling of piggyback data for `MPI_Recv`, `MPI_Irecv`, and `MPI_Wait`, respectively. All these calls use the `status` field to attach their piggyback data so that other DAMPI layers can access the data. For simplicity purposes, we omit the detail of how we attach the data to the status. Note that for the `MPI_Irecv` call, the piggyback data are not accessible until the corresponding wait is posted.

Pack_Unpack(In:buf,count,in_dtype,temp_pb; Out:out_dtype)

$lens[2] : int,\ addr[2]$: MPI_Aint, $types[2]$: MPI_Datatype
$temp\_pb \leftarrow piggyback\_buffer$
$types[0] \leftarrow$ MPI_INT
$lens[0] \leftarrow piggyback\_size$
`MPI_Address`$(temp\_pb, addr[0])$
$types[1] \leftarrow in\_dtype$
$lens[1] \leftarrow count$
`MPI_Address`$(buf, addr[1])$
`MPI_Type_struct`$(2, lens, addr, types, out\_dtype)$
`MPI_Type_commit`(out_dtype)

Figure 5.8: Packing and unpacking piggyback data - point-to-point

MPI_Send(In:buf,count,dtype,dest,tag,comm)

{Assume $temp\_buf$ is allocated on heap}
$temp\_buf$: $int[piggyback\_size]$
$packed\_dtype$: `MPI_Datatype`
`Pack_Unpack`$(buf, count, dtype, temp\_buf, packed\_dtype)$
`PMPI_Send(MPI_BOTTOM,`$1, new\_dtype, dest, tag, comm)$
`MPI_Type_free`$(packed\_dtype)$

Figure 5.9: Pseudocode for piggybacking in `MPI_Send`

MPI_Isend(In:buf,count,dtype,dest,tag,comm; Out:request)

temp_buf : *int*[*piggyback_size*]
packed_dtype : `MPI_Datatype`
`Pack_Unpack`(*buf*, *count*, *dtype*, *temp_buf*, *packed_dtype*)
`PMPI_isend(MPI_BOTTOM,`1, *new_dtype*, *dest*, *tag*, *comm*, *new_request*)
{Store *temp_buf* and *packed_type*}
`StoreTemporaryVar(..)`

Figure 5.10: Pseudocode for piggybacking in `MPI_Isend`

MPI_Recv(In:buf,count,dtype,dest,tag,comm; Out:status)

temp_buf : *int*[*piggyback_size*]
pack_dtype : `MPI_Datatype`
`Pack_Unpack`(*buf*, *count*, *dtype*, *temp_buf*, *new_dtype*)
`PMPI_Recv(MPI_BOTTOM,`1, *new_dtype*, *dest*, *tag*, *comm*, *status*)
`MPI_Type_free`(*packed_dtype*)
`AttachPBToStatus`(*temp_buf*, *status*)

Figure 5.11: Pseudocode for piggybacking in `MPI_Recv`

MPI_Irecv(In:buf,count,dtype,dest,tag,comm; Out:request)

temp_buf : *int*[*piggyback_size*]
pack_dtype : `MPI_Datatype`
`Pack_Unpack`(*buf*, *count*, *dtype*, *temp_buf*, *new_dtype*)
`PMPI_Irecv(MPI_BOTTOM,`1, *new_dtype*, *dest*, *tag*, *comm*, *request*)
`StoreTemporaryVar(..)`

Figure 5.12: Pseudocode for piggybacking in `MPI_Irecv`

MPI_Wait(In:request; Out:request,status)

if (Request is a Send) **then**
  `PMPI_Wait`(*request*, *status*)
  {Free temporary buffer and datatype stored earlier}
  `FreeTemporaryVar(..)`
else
  `PMPI_Wait`(*request*, *status*)
  `AttachPBToStatus(..)`
  {Free temporary buffer and datatype stored earlier}
  `FreeTemporaryVar(..)`
end if

Figure 5.13: Pseudocode for piggybacking in `MPI_Wait`

5.3.2.2 DAMPI Driver

The driver is the central component of DAMPI. It is responsible for maintaining the logical clocks of the process and keeping track of the $\xrightarrow{\text{mb}}$ relationship between MPI calls. The core function of the driver is to detect alternative matches for nondeterministic receives. Each driver (LLCP or LVCP) passes and receives the logical clocks to and from the piggyback module, respectively. Since the driver module and the piggyback module work tightly together, the LLCP driver module must be used together with the LLCP piggyback module. The situation is similar for the case of LVCP. Incorrect pairing of the driver module and the piggyback module may result in undefined behavior. At the end of the execution, all information necessary for the scheduler to determine whether replays are necessary is written into a database. In the current version of DAMPI, we choose to gather all data to a particular node (*MASTER_NODE*) and output the data as a single file. During replay, the driver reads in a decision database output earlier by a replay scheduler and enforces wildcard matching based on the database. Currently, the driver checks whether it is in replay mode (*GUIDED_MODE*) by either detecting the presence of the decision database or checking an environment variable's value.

We describe in detail how the driver implements LLCP to detect potential matches by walking through the pseudocode of `MPI_Irecv` and `MPI_Wait` (the pseudocode for `MPI_Recv` is also provided for reference). We shall use Figure 5.14, which contains the skeleton of a simple MPI program, to explain various concepts in our implementation. Figure 5.14 has two special columns on the left that provide the clocks and the associated event numbers of the wildcard receives. The clock information reflects the final clock assignment (i.e., after execution completes) while the event number indicates the order of the wildcard receives with respect to other wildcard receives issued in the same process. For example, event number 4 indicates that this wildcard receive is the 4^{th} wildcard receive

eventNo	*clock*	P_0	P_1	P_2
0	0	`irecv(*,tag=2,`h_0`)`	`send(0,tag=2)`	`send(0,tag=3)`
1	2	`irecv(*,tag=3,`h_1`)`	`send(0,tag=3)`	`send(0,tag=2)`
2	1	`recv(*,tag=2)`		
3	3	`recv(*,tag=3)`		
		`wait(`h_0`)`		
		`wait(`h_1`)`		

Figure 5.14: Wildcard receives with associated clocks

issued in this process. Since P_0 is the only process that is issuing wildcard receives in this example, all clocks and event numbers pertain to the wildcard receives in P_0 only.

Upon invoking the `MPI_Irecv` call (Figure 5.15), the process checks whether it should process this call under replay mode or not. Normally, all processes start in *SELF_RUN* mode during the first interleaving and run in *GUIDED_RUN* under subsequent replays. However, the scheduler only enforces the replay up until some certain point (which

MPI_Irecv(In:buf,count,dtype,src,tag,comm; Out:request)

```
1: if curr_clock > last_guided_clock then
2:   running_mode ← SELF_RUN
3: end if
4: if src =MPI_ANY_SOURCE then
5:   if running_mode = GUIDED_RUN then
6:     {Read decision database to know who to receive from}
7:     temp_src ← forced_map[eventNo]
8:     if temp_src ≠ NULL then
9:       PMPI_Irecv(buf, count, dtype, temp_src, tag, comm, request)
10:    else
11:      {This means this Irecv's clock is beyond our last guided clock}
12:      PMPI_Irecv(buf, count, dtype, src, tag, comm, request)
13:      event_list.add(eventNo)
14:    end if
15:    {Ending clock is MAX for Irecv, src will be updated later}
16:    RecordEvent(eventNo, curr_clock, MAX_CLOCK, count, dtype,
17:                src, tag, comm, 0)
18:  else {SELF_RUN}
19:    PMPI_Recv(buf, count, dtype, src, tag, comm, status)
20:    event_list.add(eventNo)
21:    RecordEvent(eventNo, curr_clock, MAX_CLOCK, count, dtype,
22:                status.SOURCE, tag, comm, 0)
23:  end if
24:  Pending.add(eventNo)
25:  request2clock[request] ← curr_clock
26:  request2event[request] ← eventNo
27:  eventNo ← eventNo + 1
28: else {Deterministic Irecv}
29:  PMPI_Recv(buf, count, dtype, src, tag, comm, status)
30:  request2clock[request] ← curr_clock
31:  request2event[request] ← eventNo
32: end if
```

Figure 5.15: Pseudocode for `MPI_Irecv`

is denoted as the *last_guided_clock* in the pseudocode) due to its Depth-First-Search algorithm. That is, for the first replay, the scheduler will set *last_guided_clock* to the largest clock value recorded and force the wildcard receive associated with that clock to match with a different sender while forcing all other wildcard receives to match the same senders that they match with during the previous run. Assuming the execution corresponding to the alternate matching does not result in any new interleavings, the scheduler will now set *last_guided_epoch* to the next highest clock and repeat the process. We describe the scheduler in detail in Section 5.3.2.4.

If the receive is a wildcard receive and it is running under *GUIDED_RUN*, the process reads the decision database output by the scheduler to determine with which process it should match (line 7). Since a nonblocking receive can attain the matched state at various points from the issuance point to the wait posting point, it is possible for a nonblocking call associated with a clock smaller than *last_guided_clock* to have a larger final clock. For example, consider the particular replay of the program in Figure 5.14 where the scheduler is trying to enforce a different matching for the third receive (i.e., `recv(*,tag=2)`). The second receive (i.e., `irecv(*,tag=3,`h_1`)`) is issued with a clock value of 0 but has a final clock value of 2, which is larger than the value of *last_guided_clock*, which is 1. Thus, the second receive will not have its match recorded in the decision database and the search would return *NULL* (line 10). In contrast, the first receive has a final clock value of 0, which is smaller than *last_guided_clock* and should have its matching sender recorded in the decision database (line 8). In the case where the process is running under *SELF_RUN*, the information associated with the receive and other bookkeeping data are recorded (line 21-27).

When the wait for a nonblocking receive completes (Figure 5.16), the following actions are taken if the request was for a wildcard receive:

- Update the source field in the event database (line 5)
- Complete all pending receives that have matched before this one according to rule $R4$ of LLCP (line 9). The procedure *CompleteNow* handles this task. We have already explained in detail this concept earlier in the discussion of LLCP in Section 5.1. The pseudocode for this procedure is provided in Figure 5.17 for reference.
- Update the final clock of the receive if necessary.

Finally, the wait extracts the piggybacked clock from the incoming message and uses the clock information to determine whether the incoming message can be a potential match

MPI_Wait(In:request; Out:request,status)

```
1: PMPI_Wait(request, status)
2: if Request is a receive then
3:   if Request is for ANY_SOURCE then
4:     this_event ← request2event[request]
5:     eventmap[this_event].src ← status.MPI_SOURCE
6:     {Can't have potential matches from the same process}
7:     eventmap[this_event].potential_matches.remove(status.MPI_SOURCE)
8:     CompleteNow(..)
9:     {Update the clock if necessary}
10:    if eventmap[this_event].end_clock = MAX_CLOCK then
11:      eventmap[this_event].end_clock ← curr_clock
12:      curr_clock ← curr_clock + 1
13:    end if
14:  end if
15:  {The Piggyback module attached piggyback data to status}
16:  incoming ← GetPiggybackFromStatus(status)
17:  ProcessIncomingMessage(..)
18: end if
```

Figure 5.16: Pseudocode for `MPI_Wait`

CompleteNow(In:status,comm,eventNo)

```
1: this_event ← eventmap[eventNo]
2: for i = 0 to Pending.size() do
3:   {Get all receives that matched before this event}
4:   if eventmap[Pending[i]] --mb--> this_event then
5:     eventmap[Pending[i]].end_clock ← curr_clock
6:     curr_clock ← curr_clock + 1
7:     Pending.remove(i)
8:   end if
9: end for
```

Figure 5.17: Pseudocode for CompleteNow

to previously issued receives.

Figure 5.18 describes the algorithm for the `MPI_Recv` call, which is similar to the combined effect of `irecv` and `wait`.

Figure 5.19 provides the pseudocode for the procedure *ProcessIncomingMessage*, which processes an incoming message to determine if it is a potential match for other receives. The procedure begins by inspecting the list of all pending receives to see if the incoming

MPI_Recv(In:buf,count,dtype,src,tag,comm; Out:status)

```
if curr_clock > last_guided_clock then
  running_mode ← SELF_RUN
end if
if src =MPI_ANY_SOURCE then
  if running_mode = GUIDED_RUN then
    {Read decision database to know who to receive from}
    src = forced_map[eventNo]
    PMPI_Recv(buf, count, dtype, src, tag, comm, status)
    {Complete the receives that matched earlier due to nonovertaking}
    CompletePendingIrecv(status, comm, eventNo)
  else
    PMPI_Recv(buf, count, dtype, src, tag, comm, status)
    CompletePendingIrecv(status, comm, eventNo)
    event_list.add(eventNo)
    RecordEvent(eventNo, curr_clock, count, dtype,
                status.SOURCE, tag, comm, 0)
  end if
  eventNo ← eventNo + 1
  curr_clock ← curr_clock + 1
else
  PMPI_Recv(buf, count, dtype, src, tag, comm, status)
  CompletePendingIrecv(status, comm, eventNo)
end if
```

Figure 5.18: Pseudocode for `MPI_Recv`

ProcessIncomingMessage(In:eventNo,incoming_clock,request,status,comm)

```
1: {Check the pending receives for possible matches}
2: for i = 0 to Pending.size() do
3:    e ← eventmap[Pending[i]]
4:    if eventNo > Pending[i] ∧ e.comm = comm ∧ (e.tag = status.MPI_TAG ∨ e.tag =
      MPI_ANY_TAG) then
5:       {If it did not match earlier, it is eligible}
6:       if e.src ≠ status.MPI_SOURCE then
7:          e.potential_matches.add(status.MPI_SOURCE)
8:          {Exit after the first match due to nonovertaking}
9:          break
10:      end if
11:   end if
12: end for
13: {Update the clock if necessary}
14: if curr_clock ≤ incoming_clock then
15:   curr_clock ← incoming_clock
16: else
17:   {This one might be a late message}
18:   {Case: blocking recv}
19:   if request = NULL then
20:      posted_clock ← curr_clock − 1
21:      {Look between posted_clock to incoming_clock for matches}
22:   else
23:      {Case: nonblocking recv}
24:      {Try to find out when this irecv posted}
25:      if eventNo ∈ event_list then
26:         {Wildcard case}
27:         posted_clock ← eventmap[eventNo].end_clock
28:      else
29:         {Deterministic case}
30:         posted_clock ← request2clock[request]
31:      end if
32:   end if
33:   FindPotentialMatches(status.MPI_SOURCE, status.MPI_TAG, comm,
34:                        posted_clock, incoming_clock, eventNo)
35: end if
```

Figure 5.19: Pseudocode for ProcessIncomingMessage

message can match any of them. If so, the procedure adds the sender of the message to the list of possible matches (line 2-12). Then, it updates the process' current clock if the piggybacked clock has a higher value. Otherwise, if the piggybacked clock is less than the current clock, we further process the message to determine if it is a late message. Recall from Theorem 5.3 that a message matching r' is a potential match to a receive r issued before r' if it has a clock smaller than or equal to r' clock (which is *posted_clock* in the pseudocode) and less than r clock. Thus, the procedure only inspects those events whose final clocks are between the message clock and r' clock (line 19-34). The detailed check for a message's eligibility (as a potential match) is performed by the procedure *FindPotentialMatches*. The logic behind this procedure is self-explanatory and thus omitted. We provide the pseudocode for this procedure in Figure 5.20 for reference.

We now describe how DAMPI handles probe calls. In an MPI execution, the processes can check for the presence of incoming messages and their associated characteristics without actually receiving them through probing operations. Probing is useful in situations where the users do not know the size of the incoming messages in advance and do not want to risk underallocating the receiving buffer, which might result in an error. Another

FindPotentialMatches(In:src,tag,comm,posted_clock,incoming_clock,eventNo)

```
1: for i = 0 to event_list.size() do
2:    e ← eventmap[event_list[i]]
3:    flag ← true
4:    flag ← flag ∧ e.end_clock ≥ incoming_clock
5:    flag ← flag ∧ e.end_clock < posted_clock
6:    flag ← flag ∧ e.end_clock > last_guided_clock
7:    flag ← flag ∧ e.end_clock ≠ MAX_CLOCK
8:    flag ← flag ∧ event_list[i] ≤ eventNo
9:    flag ← flag ∧ (e.tag = tag||e.tag = MPI_ANY_TAG)
10:   flag ← flag ∧ (e.comm = comm)
11:   if flag then
12:      if src ≠ e.src ∧ src ∉ e.potential_matches then
13:         e.potential_matches.add(src)
14:      end if
15:   else if e.end_clock < incoming_clock ∨ e.end_clock <= last_guided_clock then
16:      break
17:   end if
18: end for
```

Figure 5.20: Pseudocode for FindPotentialMatches

option is to overallocate the receiving buffer but this might be undesirable in systems with memory constraints.

MPI supports two different methods of probing and we describe how DAMPI handles them.

- `MPI_Probe` is a blocking probe, which behaves similarly to `MPI_Recv` in the sense that it blocks until there is at least a matching incoming message. Further, `MPI_Probe` can also use `MPI_ANY_SOURCE` as its source argument and thus can be a source of nondeterminism. DAMPI handles `MPI_Probe` similarly to `MPI_Recv` with several minor differences as follows. First of all, `MPI_Probe` does not actually receive the message and thus there is no piggybacked data to deal with. Second, the successful completion of `MPI_Probe` only indicates that there is at least one message. Therefore, DAMPI only updates the first pending receive in the $CompleteNow$ set. Figure 5.21 provides the updated pseudocode of the procedure `CompleteNow` with support for probing.
- `MPI_Iprobe` is a nonblocking probe which returns a boolean *flag* to indicate whether there is a message to receive or not. This is in contrast with `MPI_Probe`, which returns only when there is a message to receive. DAMPI only processes `MPI_Iprobe` when the variable *flag* is set to true, in which case it is treated similarly to `MPI_Probe`.

We briefly describe how DAMPI handles `MPI_ANY_TAG` since the usage of `MPI_ANY_TAG` creates another difficulty with respect to nondeterminism coverage. Consider the example in Figure 5.22 in which the second nonblocking wildcard receive from P_0 uses `MPI_ANY_TAG`,

CompleteNow(In:status,comm,eventNo,isProbe)

```
1: this_event ← eventmap[eventNo]
2: for i = 0 to Pending.size() do
3:    {Get all receives that matched before this event}
4:    if eventmap[Pending[i]] --mb--> this_event then
5:       eventmap[Pending[i]].end_clock ← curr_clock
6:       curr_clock ← curr_clock + 1
7:       Pending.remove(i)
8:       if isProbe then
9:          break
10:      end if
11:   end if
12: end for
```

Figure 5.21: Pseudocode for CompleteNow with probe support

P_0	P_1	P_2
`irecv(*,tag=3,`h_0`)`	`send(0,tag=3)`	`recv(0,tag=100)`
`irecv(*,tag=*,`h_1`)`	`send(0,tag=3)`	`send(0,tag=3)`
`recv(*,tag=2)`	`send(0,tag=2)`	
`send(2,tag=100)`		
`wait(`h_0`)`		
`wait(`h_1`)`		

Figure 5.22: MPI example with MPI_ANY_TAG

which is denoted as `*`. Manual inspection allows us to conclude that this program is actually deterministic; that is, each wildcard receive has exactly one possible match. However, without special handling of `MPI_ANY_TAG`, both LLCP and LVCP will incorrectly deduce that the call `send(0,tag=3)` from P_2 is a potential match to the first wildcard receive of P_0. The reason is that at the point of completing the blocking wildcard receive in P_0, we do not know the exact tag of the message received by the `irecv(*,tag=*,`h_1`)` call from P_0. Without the tag information, the algorithm cannot determine whether the `irecv(*,tag=3,`h_0`)` call is required to match before the call with `MPI_ANY_TAG`. Therefore, we need a mechanism to obtain the tag of a nonblocking receive using `MPI_ANY_TAG` when the algorithms determine that such a receive has already attained the matched state. We address this problem by using the operation `MPI_Request_get_status`, which allows us to obtain the status of a nonblocking call without destroying its request handle. Once we determine that a nonblocking receive using `MPI_ANY_TAG` has matched, we simply do a busy-wait loop with `MPI_Request_get_status` to obtain the status, which allows us to figure out the tag information.

5.3.2.3 Error Checking Modules

DAMPI provides several error checking modules that provide correctness checks for deadlock and resource leaks. We briefly summarize these modules here.

- Deadlock detection: DAMPI provides a lightweight timeout-based deadlock detection module. During the initialization phase (`MPI_Init`), each process spawns a new thread that communicates with other threads from other processes to determine when the processes have entered a deadlock scenario. The threads use MPI to communicate themselves and thus the module requires `MPI_THREAD_MULTIPLE` support from the MPI runtime to operate. Since we only focus on MPI-related deadlock,

the threads conclude that the execution has deadlocked if and only if all processes have not returned from some MPI calls, and they also have not progressed since the last time the threads synchronized. The time interval between two successive synchronization points can be fine-tuned by the user. In our experiments, this simple scheme is very effective in practice and scalable. If absolute soundness guarantee is required for deadlock detection, a more precise scheme such as the Wait-For-Graph approach [32] should be adapted. However, such approach would not scale as well as our proposed scheme.

- Resource leaks: We use a simple counting mechanism to keep track of nonblocking requests and user-defined communicators to detect whether all requests have been finished and all communicators have been freed.

5.3.2.4 The DAMPI Scheduler

We implement the scheduler based on the concept of the ISP scheduler [63]. The main difference is that the DAMPI scheduler operates postmortem and does not interact with the processes while they execute. Thus, it is very scalable and easy to parallelize. Figure 5.23 provides the pseudocode for the scheduler. For simplicity, we omit the pseudocode of several auxiliary procedures and only provide the pseudocode for the main procedure, `ExploreInterleavings`. The functionalities of the auxiliary procedures are summarized here.

- `parseProcessOutput` reads in the output from the processes and stores the clock information as well as the last clock value, which is the largest clock value recorded by all processes. Note that since `MPI_Finalize` acts as a synchronization point for all processes, this largest clock value should be the same for all processes. The last clock value is denoted as *last_clock*. The scheduler stores each clock value as a mapping between the clock and its possible outcomes. Consider the situation where a wildcard receive carrying an event number e from P_0 matches with a message sent from P_1 in clock c and lists P_2 as a potential match. The scheduler stores such information as $\{c \rightarrow (e, [1, 2])\}$. The first number in the bracket represents the match that the process observes during the last execution while the rest are the potential matches. Note that the processes only record those wildcard receives that have at least one potential match. `parseProcessOutput` returns `true` if there are potential matches discovered during the execution, otherwise it returns `false`.

ExploreInterleavings()

```
1: if parseProcessOutput() = false then
2:   {There is only one interleaving possible}
3:   return
4: end if
5: last_guided_clock ← last_clock − 1
6: while last_guided_clock ≥ 0 do
7:   if hasPotentialMatches(last_guided_clock) then
8:     for all m ∈ getMatches(last_guided_clock) do
9:       interleavings[last_guided_clock].push(m)
10:    end for
11:    foundExtraMatches ← false
12:    while interleavings[last_guided_clock].size() ≠ 0 do
13:      outputToDecisionDatabase(interleavings[last_guided_clock].pop())
14:      restartTheProcess()
15:      {See if there are new matches}
16:      if parseProcessOutput() then
17:        foundExtraMatches ← true
18:        {DFS - Need to process new matches first}
19:        break
20:      end if
21:    end while
22:    if foundExtraMatches then
23:      last_guided_clock ← last_clock − 1
24:      continue
25:    end if
26:    {Done with this clock}
27:    cleanUp(last_guided_clock)
28:  end if
29:  last_guided_clock ← last_guided_clock − 1
30: end while
```

Figure 5.23: Pseudocode for the DAMPI scheduler

- `hasPotentialMatches` returns *true* if the given clock is associated with a wildcard receive that has at least one potential match.
- `getMatches` extracts the mapping mentioned earlier to build all possible matching for a wildcard receive and returns a list of matching pairs. Each matching pair has a wildcard receive and the potential match that should be forced during the next run.
- `outputToDecisionDatabase` outputs all information necessary to replay up to the given clock number. For example, to force a receive with a clock value of 5 to match

with a different sender, it is necessary to force all previous receives for which their clocks are smaller than 5 to match with the same senders that they match with during the previous interleaving.

- `restartProcess` restarts the processes.
- `cleanUp` removes the information pertaining to a particular clock value to release the memory back to the OS.

We now describe scheduler (Figure 5.23). When the scheduler is first invoked, it parses the output from the processes to determine whether replays are necessary. If the processes do not detect any potential matches to any wildcard receives (or there are no wildcard receives), no further replay is necessary and the scheduler terminates (line 1-3). On the other hand, if there are potential matches for wildcard receives detected, the procedure `parseProcessOutput` stores all clock values and the information pertaining to the wildcard receives associated with those clocks, which include the event numbers, the senders with which the receives match, and the potential senders that they can match. Then, the scheduler starts by processing the data associated with each clock value, starting with the last clock value, to construct the possible matching for that clock value. Each possible matching will result in a new interleaving and is pushed to *interleavings* (line 9), which is a mapping between a clock value and all possible interleavings associated with that clock. Once all interleavings for a given clock are stored in *interleavings*, each possible matching is written to a decision database (line 13), which can be either a file or a relational database (currently we use files). Upon restart, the processes will force all matches according to the decision database, up to the value of *last_guided_clock* (line 14). If the processes discover more potential matches during replays, the scheduler will process those new matches and force new replays to explore them (line 16-25). Finally, the scheduler terminates when it has explored all possible interleavings.

5.3.3 Evaluation of LLCP and LVCP

We evaluate the performance of the two protocols that DAMPI provides (LLCP and LVCP). In particular, we are interested in the scalability and the accuracy of the protocols. To evaluate the scalability, we link the driver module and the piggyback module for LLCP and LVCP, respectively, to DAMPI and apply DAMPI to several benchmarks to measure the bandwidth, latency, and slowdown. Since we have already proved that LVCP is sound and complete, we evaluate the accuracy of LLCP by manual comparison of the results of LLCP to LVCP to determine if both protocols discover the same set of matches.

5.3.3.1 Experiment Setup

We run our benchmarks on the Atlas cluster available at Lawrence Livermore National Laboratory, which is a Linux-based cluster with 1152 compute nodes, 8 cores, and 16GB of memory per node. All experiments were compiled and run under MVAPICH2-1.5 [9] with 8 tasks per node.

First, we report the latency and bandwidth impact of the two protocols. For latency testing, we use the OSU multipair latency benchmark [9] and report the latency result for 4-byte messages as the number of processes increases. These small messages represent the class of messages where latency impact is most significant. For bandwidth testing, we use a simple ping-pong test between two processes in the system, while others sit idle. While typical bandwidth tests report the bandwidth as the message size grows, such tests do not take into account the number of processes in the system and thus do not provide a good way to judge the impact of the protocols. Thus, we report the message sizes where the system achieves half of the peak bandwidth for the bandwidth testing (the R/2 value).

Figures 5.24 and 5.25 summarize our bandwidth and latency testing results, re-

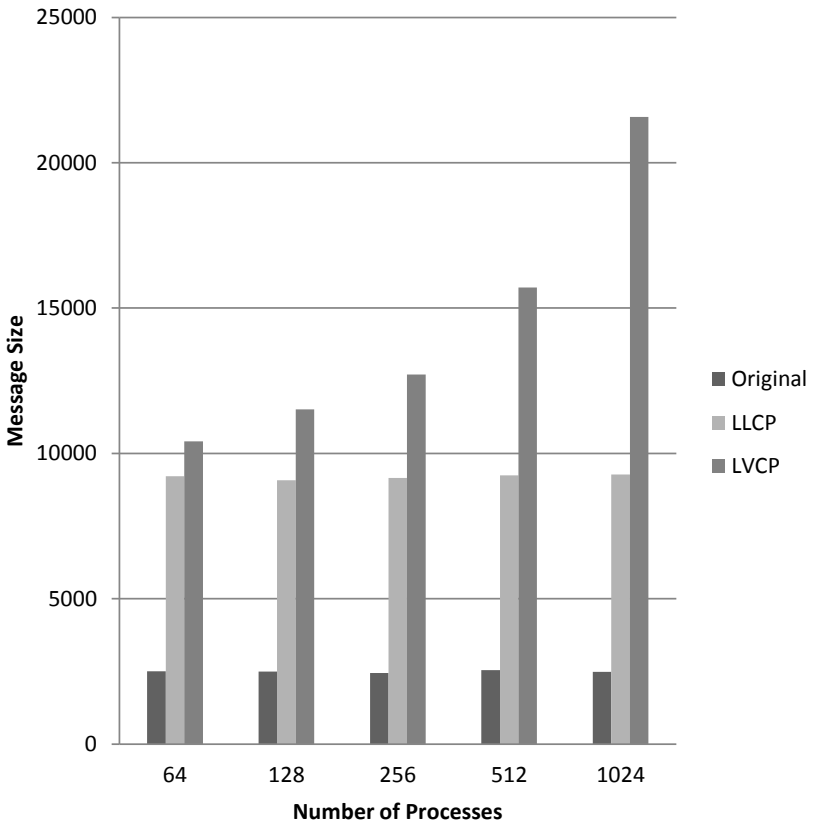

Figure 5.24: Bandwidth impact

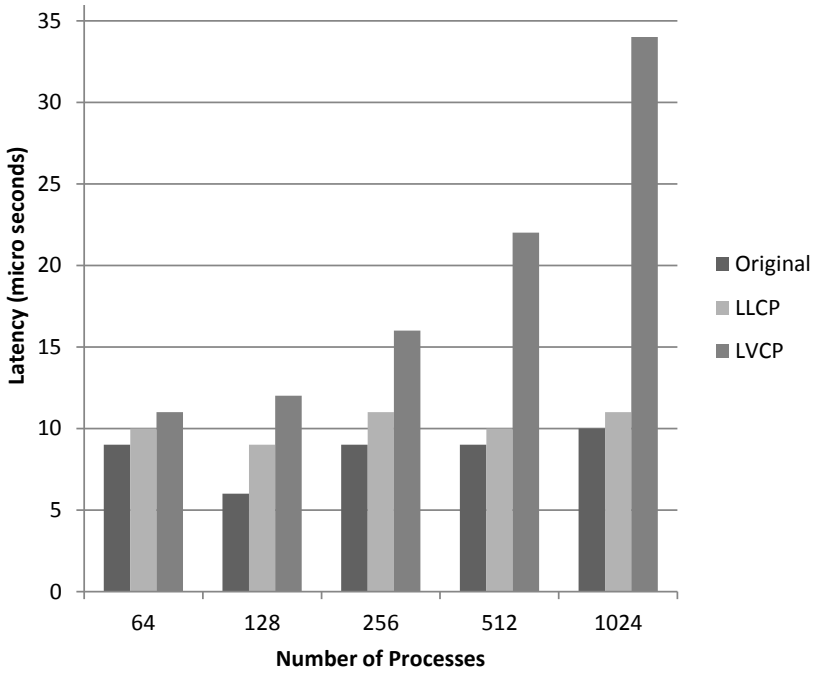

Figure 5.25: Latency impact

spectively. Note that higher bars denote worse performance in our graphs since they correspond to longer latencies and to larger message sizes to achieve the same bandwidth.

The results show that both protocols have manageable latency at lower process counts but the impact of vector clocks becomes much more significant as the system scales up while LLCP maintains nearly constant latency penalties throughout the entire range. At 1024 processes, the latency of messages under LLCP increases by only 10% compared to the original, uninstrumented messaging while it increases 240% under LVCP. Both protocols achieve essentially the same peak bandwidth as uninstrumented messaging but reduce the bandwidth achieved at intermediate message sizes. Importantly, this impact is more pronounced with LVCP and increases with increasing process count, while the impact of LLCP is independent of the process count.

Latency and bandwidth do not always translate to overhead since programs typically do not spend 100% of their CPU time exchanging messages. Therefore, we evaluate the performance of three scientific MPI applications: ParMETIS, a parallel hypergraph partitioning library [12]; AMG2006, an algebraic multigrid solver for unstructured mesh available from the ASC Sequoia benchmark [13]; and SMG2000, a semicoarsening multigrid solver from the ASCII Purple benchmark [15]. All of these applications are designed to run at very large scale. We run ParMETIS using the supplied testing parameters, both

AMG2006 and SMG2000 with a $6 \times 6 \times 6$ grid as input, with the number of processes ranging from 64 to 1024 (8 processes per node). We report the average result from given runs. ParMETIS and SMG2000 do not have any nondeterministic receives and thus the results reflect the overhead of checking for common MPI errors (e.g., resource leaks). For AMG2006, which has both wildcard probes as well as wildcard receives, we report the performance for the first run, which reflects the overhead of tracking of wildcard events and checking for errors. The results of AMG2006 under LLCP and LVCP are identical even though AMG2006 has multiple processes issuing wildcard receives, which reiterates that for most practical applications, omissions do not occur under LLCP.

Figures 5.26 and 5.27 summarize the overhead evaluation for SMG2000 and AMG2006, which display similar trends where LVCP remains competitive with LLCP until the system goes to 1024 processes, and we expect the overhead to become worse as the size of the vector clocks grows. We also notice several interesting cases where LLCP has

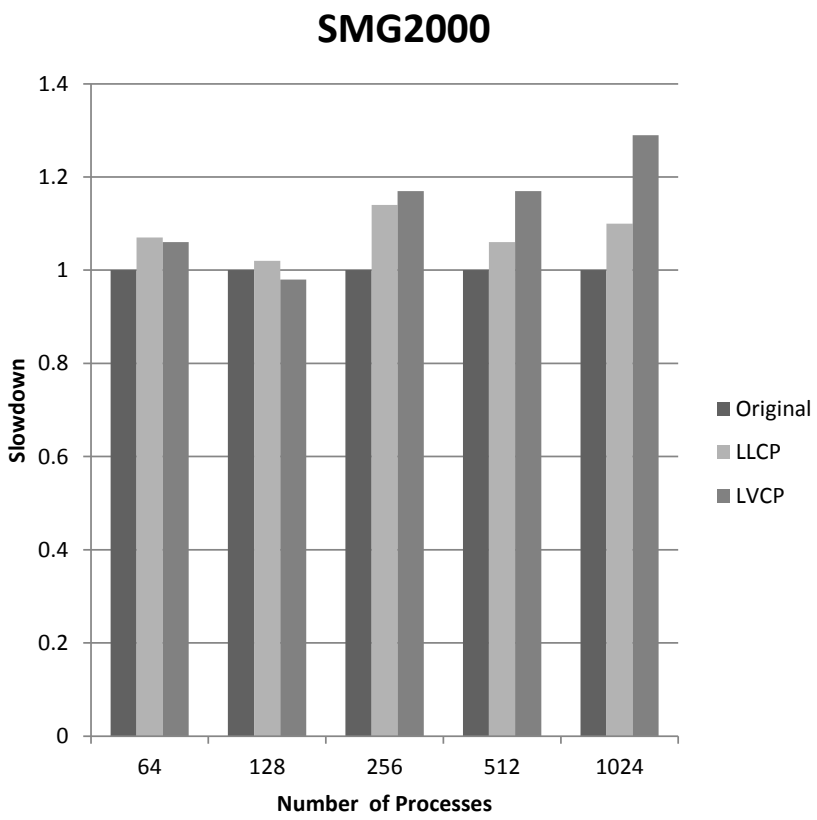

Figure 5.26: Overhead on SMG2000

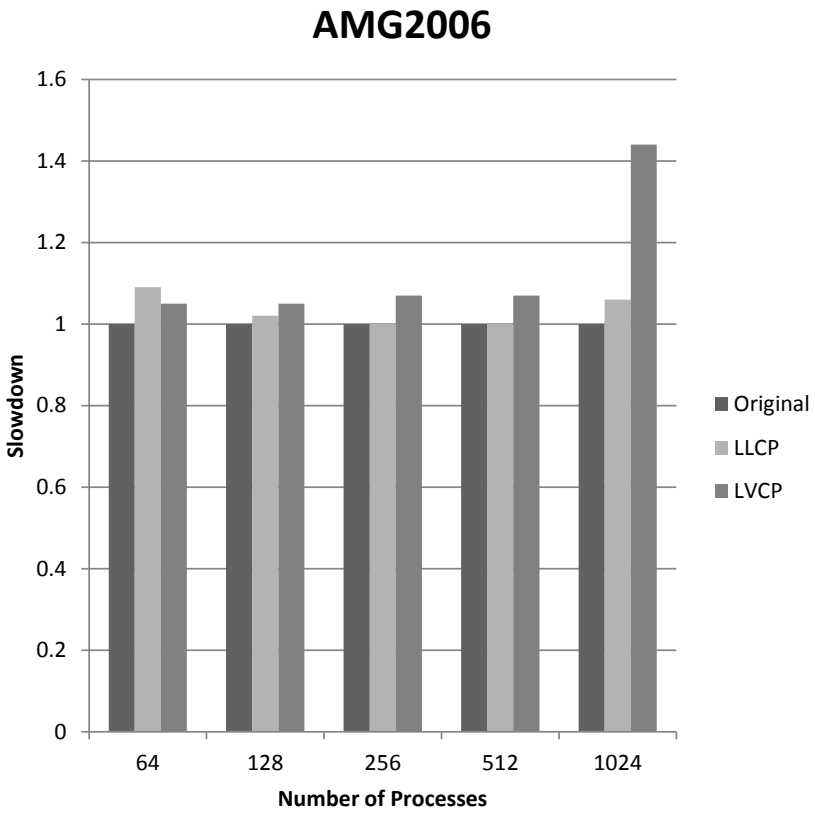

Figure 5.27: Overhead on AMG2006

negative overhead (i.e., it runs faster with extra processing), probably due to the extra payload affecting the communication patterns and resulting in better optimization from the MPI runtime.

Figure 5.28 provides the overhead evaluation for ParMETIS, which issues almost 550 million messages (compared to AMG at 200 million and SMG2000 at 40 million) under our experiment with 1024 processes; over 98% of these messages are between 4-256 bytes and thus, the verification suffers a high latency penalty caused by the transmission of the vector clocks.

One issue worth noting but not conveyed from the figures is the memory overhead associated with LVCP. For each relevant event, a vector clock must be kept in order to track the causality of the event, which results in a very large amount of memory being used for bookkeeping, especially when the programs have many wildcards and run at large scale. For example, AMG2006 generated about 1800 wildcard events per process (1024 processes total), which results in about 7 Gigabytes of extra memory (collectively) to

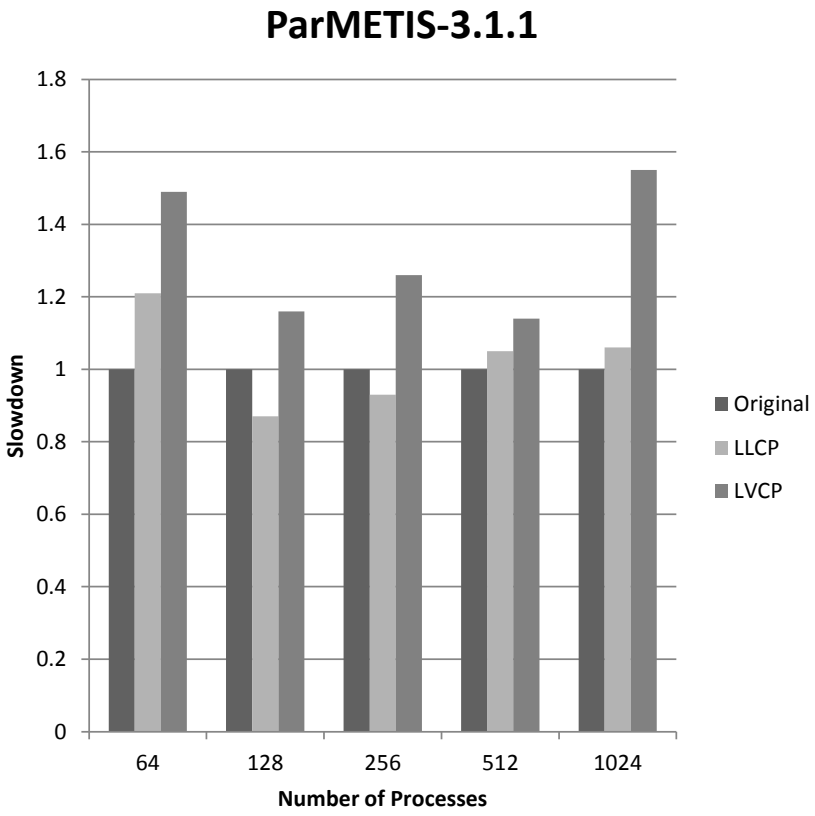

Figure 5.28: Overhead on ParMETIS-3.1.1

keep track of vector clocks. As we scale the programs to much larger scale, this memory overhead becomes prohibitively expensive.

One way to tackle the memory overhead as well as the bandwidth overhead of vector clocks is to use *compressed vector clocks* [31, 60]. Some of the proposed schemes are not directly applicable to MPI due to the special requirements on the runtime (e.g., first-in-first-out channels [60]). Nonetheless, the efficiency of these schemes are highly dependent on the communication patterns and all of them require more local memory storage (compared to traditional vector clocks) for bookkeeping. As supercomputers become larger and employ more and more cores, the amount of memory available to each core becomes smaller and the additional memory overhead might prevent the applications from running. Furthermore, all compressed vector clocks schemes require variable piggyback information being sent every time, including at collective calls. Since most MPI runtimes do not yet support native piggybacking, implementing a variable piggybacking layer forces the developer to use explicit buffer packing, which greatly increases performance overhead.

In addition, one would need to break collective calls into pair-wise in such a piggyback scheme and thus forfeit all existing MPI collective optimizations (the alternative would be to always piggyback all the vector clocks for collective).

5.3.4 DAMPI Performance Evaluation

As discussed earlier, LLCP maintains soundness and completeness in most practical situations and scales much better than LVCP. We further evaluate the performance of DAMPI compared to ISP to show that DAMPI provides comparable coverage over the space of nondeterminism while achieving greater scalability. We also experiment with different interleaving reduction heuristics which are described below.

Our evaluation uses these benchmarks:

- An MPI matrix multiplication implementation, *matmult*;
- ParMETIS-3.1 [12], a fully deterministic MPI-based hypergraph partition library;
- Benchmarks from the NAS Parallel Benchmarks (NAS-PB) 3.3 [10];
- Benchmarks from the SpecMPI2007 [14] suites; and
- The Adaptive Dynamic Load Balancing (ADLB) library [42].

Our ParMETIS, NAS-PB and SpecMPI tests measure DAMPI's overheads and target evaluation of its local error (e.g., request leaks or communicator leaks) checking capabilities. In *matmul*, we use a master-slave algorithm to compute $A \times B$. The master broadcasts the B matrix to all slaves and then divides up the rows of A into equal ranges and sends one to each slave. The master then waits (using a wildcard receive) for a slave to finish the computation. It then sends the slave another range r_j. This benchmark allows us to study the bounded mixing heuristic in detail with a well-known example. We also evaluate the bounded mixing heuristic with ADLB, a relatively new load balancing library that has significant nondeterminism and an aggressively optimized implementation. In our previous experiments using ISP, we could not handle ADLB even for the simplest of verification examples. We now discuss our results under various categories.

5.3.4.1 Full Coverage

Figure 5.29 shows the superior performance of DAMPI compared to that of ISP running with Parmetis, which makes about one million MPI calls at 32 processes. As explained earlier, due to its centralized nature, ISP's performance quickly degrades as the number of MPI calls increases, while DAMPI exhibits very low overhead. In fact, the

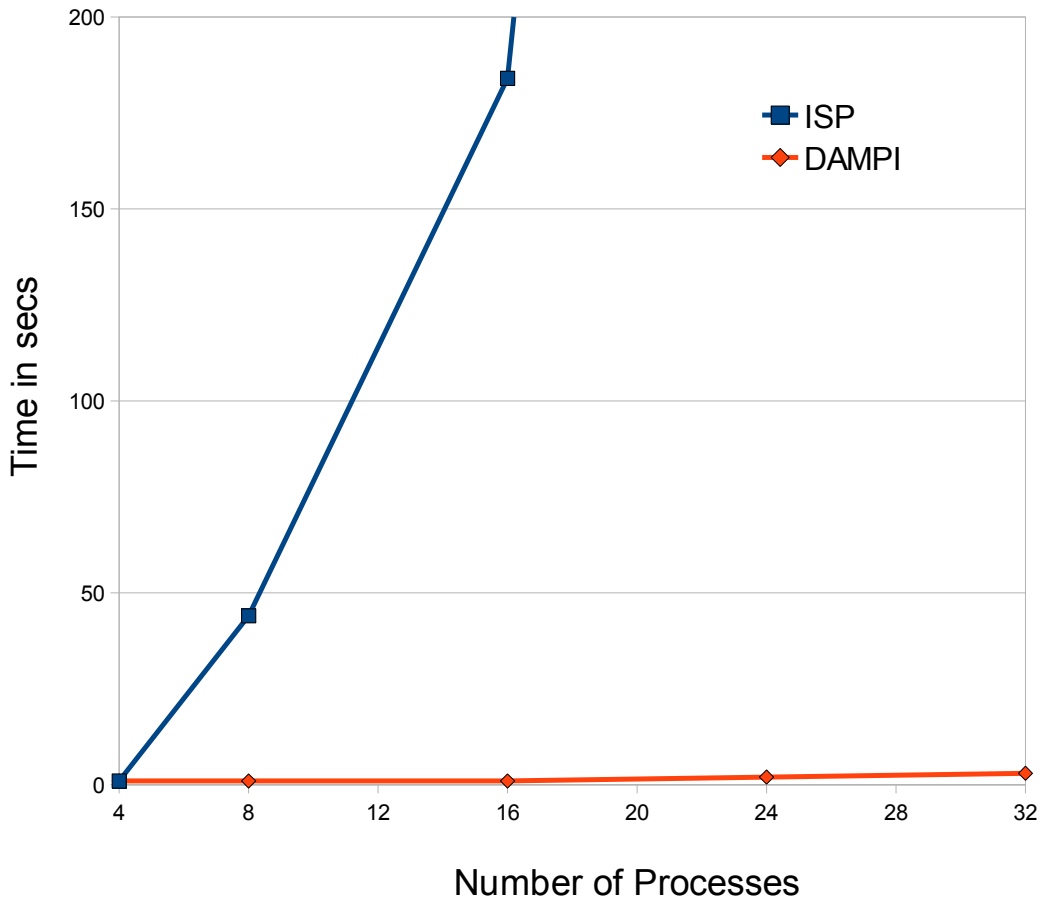

Figure 5.29: ParMETIS-3.1: DAMPI vs. ISP

overhead of DAMPI on ParMETIS is negligible until the number of processes becomes large (beyond 1K processes).

In order to understand the reasons behind the significant improvement of DAMPI over ISP better, we log all MPI communication operations that ParMETIS makes (see Table 5.1). We do not log local MPI operations such as `MPI_Type_create` or `MPI_Get_count`. We classify the operations as Send-Recv, Collective or Wait. Send-Recv includes all point-to-point MPI operations, Collective includes all collective operations, and Wait includes all variants of `MPI_Wait` (e.g., Waitall).

Although the total number of MPI operations grows by a factor of 2.5 on average as the number of the processes increases, the total number of MPI operations per process only grows by a factor of 1.3 on average. In effect, the number of MPI operations that

Table 5.1: Statistics of MPI operations in ParMETIS-3.1

MPI Operation Type	*procs=8*	16	32	64	128
All	187K	534K	1315K	3133K	7986K
All per proc.	23K	33K	41K	49K	62K
Send-Recv	121K	381K	981K	2416K	6346K
Send-Recv per proc	15K	24K	31K	38K	50K
Collective	20K	36K	63K	105K	178K
Collective per proc	2.5K	2.2K	2.0K	1.6K	1.4K
Wait	47K	118K	272K	612K	1463K
Wait per proc	5.8K	7.3K	8.5K	9.6K	11K

the ISP scheduler must handle increases almost twice as fast as the number of MPI operations that each process in DAMPI must handle as the number of processes increases (due to the DAMPI's distributed nature). Each type of MPI operation behaves similarly, especially the collective calls, for which the number of operations *per process* decreases as the number of processes increases. In addition to the increasing workload placed on the ISP scheduler, the large number of local MPI processes also stresses the system as a whole, which explains the switching from linear slowdown to exponential slowdown around 32 processes. We also experimented with the distributed version of ISP that allows the processes to be launched on different nodes but that version actually performs even worse compared to the local version ISP. The data further confirm our observation that the centralized scheduler is ISP's biggest performance bottleneck.

To evaluate the overhead of DAMPI further, we apply DAMPI on a range of medium to large benchmarks, including the NAS-NPB 3.3 suite and several codes from the SpecMPI2007 suite. We run the experiments on an 800 node, 16 cores per node Opteron Linux cluster with an InfiniBand network running MVAPICH2 [9]. Each node has 30GB of memory shared between all cores. We submit all experimental runs through the Moab batch system and use the wall clock time as reported by Moab to evaluate the performance overhead. Table 5.2 shows the overhead of running DAMPI with 1024 processes.

In Table 5.2, the R* column gives the number of wildcard receives that DAMPI analyzed while C-leak and R-leak indicate if we detected any unfreed communicators and pending requests (not completed before the call to `MPI_Finalize`).

Next we evaluate the tools' efficiency in processing the interleavings by applying the tools to *matmul* and measure how long it takes for DAMPI and ISP to explore through the possible different interleavings of *matmul*. Our experiments show that DAMPI can

Table 5.2: DAMPI overhead: Large benchmarks at 1K processes

Program	Slowdown	Total R*	C-Leak	R-Leak
ParMETIS-3.1	1.18x	0	Yes	No
104.milc	15x	51K	Yes	No
107.leslie3d	1.14x	0	No	No
113.GemsFDTD	1.13x	0	Yes	No
126.lammps	1.88x	0	No	No
130.socorro	1.25x	0	No	No
137.lu	1.04x	732	Yes	No
BT	1.28x	0	Yes	No
CG	1.09x	0	No	No
DT	1.01x	0	No	No
EP	1.02x	0	No	No
FT	1.01x	0	Yes	No
IS	1.09x	0	No	No
LU	2.22x	1K	No	No
MG	1.15x	0	No	No

offer coverage guarantees over the space of MPI nondeterminism while maintaining vastly improved scalability when compared to ISP – the current state-of-the-art dynamic formal verifier for MPI programs. We attribute this improvement in handling interleavings to the lack of synchronous communication within DAMPI. All extra communication introduced by DAMPI is done through MPI piggyback messages, which has been shown to have very low overhead [51]. Figure 5.30 summarizes our experiments.

However, naïvely approaching the exponential space of interleavings in heavily nondeterministic programs is not a productive use of verification resources. We now present several heuristics implemented in DAMPI that can allow the user to *focus* coverage to particular regions of interest, often exponentially reducing the exploration state space.

5.3.5 Search Bounding Heuristics Evaluation

Full coverage over the space of MPI nondeterminism is often infeasible, even if desirable. Consider an MPI program that issues N wildcard receives in sequence, each with P potential matching senders. Covering this program's full state space would require a verifier to explore P^N interleavings, which is impractical even for fairly small values (e.g., $P = N = 1000$). While these interleavings represent unique message matching orders, most cover the same (equivalent) state space if the matching of one wildcard receive is independent of other matches. Consider these common communication patterns:

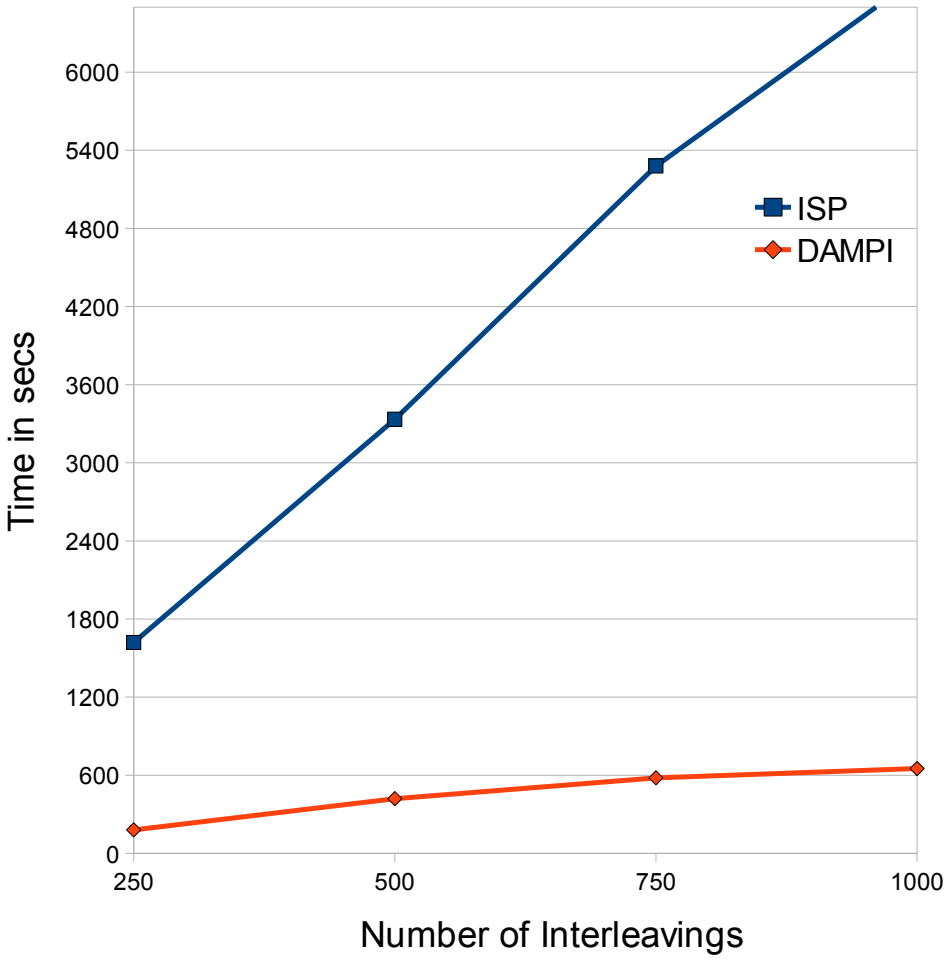

Figure 5.30: Matrix multiplication: DAMPI vs. ISP

- A master/slave computation in which the master receives the computed work from the slaves and stores it in a vector indexed by the slave's rank;
- A series of computational phases in which processes use wildcard receives to exchange data and then synchronize.

Both patterns do not require that we explore the full state space. Clearly, the order of posting the master's receives does not affect the ending state of the program. Similarly, while the order of message matching within a single phase of the second pattern might lead to different code paths within a phase, the effect is usually limited to that particular phase.

Recognizing such patterns is a challenge for a dynamic verifier such as DAMPI, which has no knowledge of the source code. Further, complicated looping patterns often make

it difficult to establish whether successive wildcard receives are issued from within a loop. Similarly, an `MPI_Allreduce` or an `MPI_Barrier` does not necessarily signal the end of a computation phase. Thus, it is valuable to capitalize on the knowledge of users who can specify regions on which to focus analysis. Such hints can significantly improve the coverage of *interesting* interleavings by a tool such as DAMPI. We now discuss our two *complementary* search bounding techniques, *loop iteration abstraction* and *bounded mixing search*.

5.3.5.1 Loop Iteration Abstraction

Many programs have loops with a fixed computation pattern that a verifier can safely ignore. By turning off interleaving exploration for nondeterministic matches occurring within such loops, DAMPI can explore other nondeterministic matches more thoroughly.

To use this feature in DAMPI, the user must insert `MPI_Pcontrol` calls at the beginning and end of loops that should not be explored. Upon logging these `MPI_Pcontrol` calls, DAMPI pursues only the matches it discovers during *SELF_RUN*, and avoids exploring alternative matches. Despite its simplicity, loop iteration abstraction can substantially reduce the iteration space that DAMPI must explore. In the future we will build static analysis based instrumentation facilities to semi-automate this heuristic.

5.3.5.2 Bounded Mixing

Many search bounding techniques exist. Bounded model checking [2] unravels the state space of a system to a finite depth. This heuristic suits hardware systems for which reachability graphs are considerably smaller than in software.

Context bounding [45] is much more practical in that it does not bias the search towards the beginning of state spaces. In effect, it runs a program under small *preemption quotas*. More specifically, special schedulers allow preemption two or three times *anywhere* in the execution. However, the scheduler can only employ a small fixed number of preemptions, after which it can switch processes only when they block.

While preemption bounding is powerful for shared memory concurrent programs based on threads, it is only marginally useful for message passing programs. In message passing, simple preemption of MPI processes is highly unlikely to expose new bugs (as explained earlier, one must take active control over their matchings). Also most preemptions of MPI programs prove useless since context-switching across deterministic MPI calls does

not reduce the state space. We have invented bounded mixing, a new bounding technique that is tailor-made to how MPI programs work.

The intuition behind bounded mixing can be explained as follows: we have observed that each process of an MPI program goes through *zones* of computation. In each zone, the process exchanges messages with other processes and then finishes the zone with a collective operation (e.g., a reduction or barrier). Many such sequential zones cascade along – all starting from `MPI_Init` and ending in `MPI_Finalize`. In many MPI programs, these zones contain wildcard receives, and cascades of wildcard receives quickly end up defining large (exponential) state spaces. Figure 5.31 depicts an abstraction of this pattern. In this figure, A is a nondeterministic operation (e.g., a wildcard receive), followed by a zone followed by a collective operation. B then starts another zone and the pattern continues. If each zone contains nondeterministic operations, then the possible interleavings is exponential in the number of zones (no interleaving explosion occurs if all zones contain only deterministic operations).

We intuitively believe that zones that are far apart usually do not interact significantly. We define the distance between two zones by the number of MPI operations between them. The intuition behind this statement is that each zone receives messages, responds, and moves along through a lossy operation (e.g., a reduction operation or a barrier). In particular, conditional statements coming later are not dependent on the computational results of zones occurring much earlier.

Based on these empirical observations above, bounded mixing limits the exploration of later zones to e.g., representative paths arriving at the zone instead of exploring *all paths* arriving at the zone. Thus, we explore the zones beginning at C only under the leftmost path A,B,C. We do not explore the zones beginning at C under all four paths. This example is actually bounded mixing with a mixing bound of $k = 2$ (the zones beginning at C and E are allowed to "mix" their states, and so do the zones beginning at C and D). We also allow the zones beginning at B and C to mix their states. Finally, we will allow the zones beginning at A and B to mix their states.

Setting mixing bounds results in search complexity that grows *only as the sum of much smaller exponentials*. Using our example program with P^N possible interleavings earlier, a $k = 0$ setting will result in $P * N$ interleavings while a $k = unbounded$ setting will result in full exploration. Bounded mixing in DAMPI provides knobs that designers can set for various regions of the program: for some zones, they can select high k values

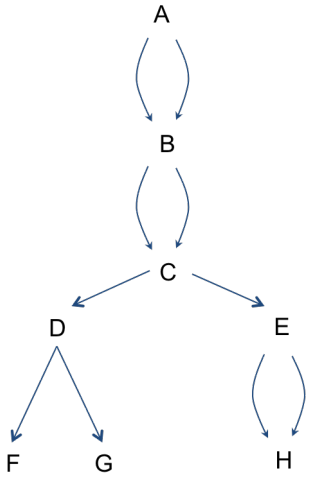

Figure 5.31: A simple program flow to demonstrate bounded mixing

while for others, they can select low values, which supports a selectively focused search.

We now briefly explain how we implemented bounded mixing in DAMPI. Suppose the search is at some clock s, and suppose s has several as yet unexplored potential matches *but all subsequent clocks of s have been explored.*

Then, the standard algorithm will: (i) pursue the unexplored option at s, and (ii) recursively explore *all paths below that option.* In bounded mixing search, we will: (i) pursue the unexplored option at s, and (ii) recursively explore *all paths below that option* up to depth k. Thus, if B's right-hand side entry has not been explored, and if $k = 2$, then we will (i) descend via the right-hand side path out of B, and (ii) go only two steps further in all possible directions. After those k steps, we simply let the MPI runtime determine wildcard receive matching.

To evaluate bounded mixing, we first show the effects of bounded mixing on our small and simple application: *matmul.* Figure 5.32 shows the results of applying several different values of k. As expected, bounded mixing greatly reduces the number of interleavings that DAMPI explores. However, our heuristic has another subtle yet powerful advantage: the number of interleavings increases in a linear fashion when k increases. Thus, users can slowly increase k should they suspect that the reaching effect of a matching receive is further than they initially assumed.

We also apply bounded mixing to the Asynchronous Dynamic Load Balancing (ADLB) library [42]. As the name suggests, ADLB is a highly configurable library that can run with a large number of processes. However, due to its highly dynamic nature, the degree

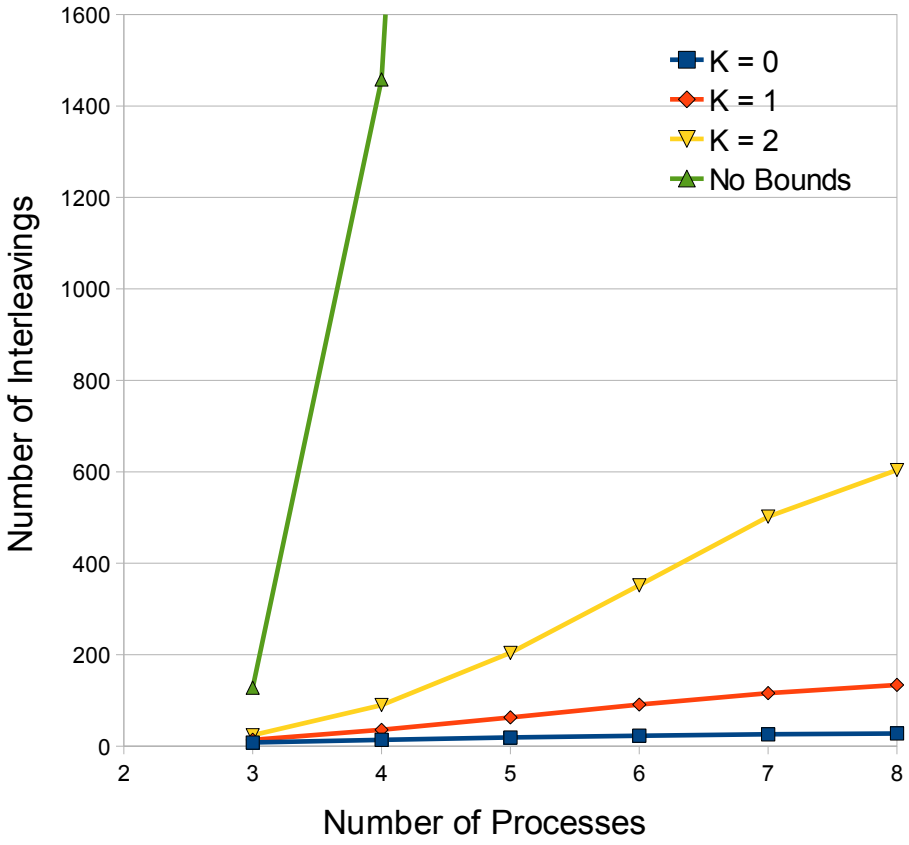

Figure 5.32: Matrix multiplication with bounded mixing applied

of nondeterminism of ADLB is usually far beyond that of a typical MPI program. In fact, verifying ADLB for a dozen processes is already impractical, let alone for the scale at which DAMPI targets. Figure 5.33 shows very encouraging results of verifying ADLB with various values of k.

In summary, bounded mixing is a promising scheme to reduce the search space by prioritized replaying the executions in which nondeterminism has bounded impact. As we have shown, different values of the bounding factor k have great impact on the number of interleavings explored. Each MPI application likely requires a different k to achieve the required coverage, which we plan on exploring as our future work.

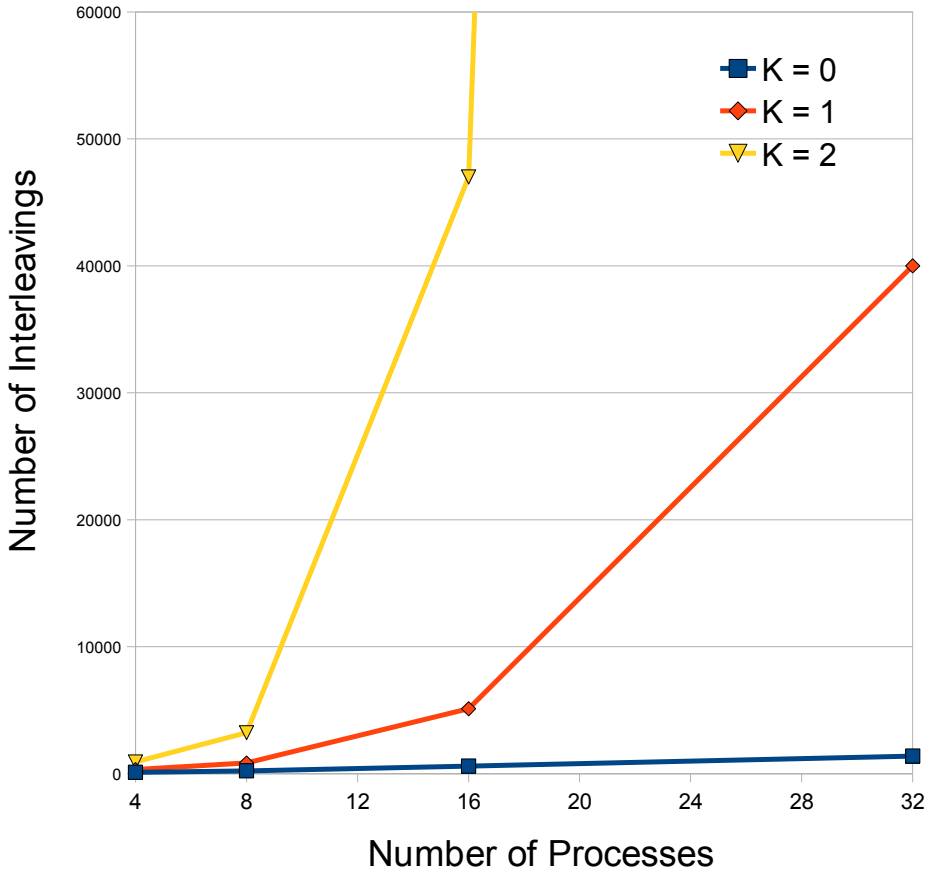

Figure 5.33: ADLB with bounded mixing applied

CHAPTER 6

RELATED WORK

Discovering bugs in parallel programs is a very challenging task. The gigantic code size and large degree of complexity of most real-life parallel applications usually render conventional debugging methods such as `gdb` [4] or `printf` impractical. Fortunately, there has been an enormous amount of research effort spent on creating tools specialized in debugging these kinds of applications, largely due to the increased availability and popularity of large clusters. We provide a quick summary of these tools and their functionalities. We group them into three major categories: MPI testing and debugging tools, MPI verification tools, and deterministic replay tools. Note that several works might fit into more than one category, but we will nonetheless attempt to categorize them based on their most popular usages.

6.1 Debugging and Correctness Checking

This category includes all MPI debuggers and correctness checking tools. We further split this category up into debuggers and correctness checking tools. Debuggers only provide a debugging interface and do not offer any error checking capabilities while most correctness checking tools do not allow the developers to interact with the MPI processes while they are running.

6.1.1 Debugging

Tools like DDT [1] and Totalview [17, 27] are often regarded as the gdb for MPI programs. In fact, DDT attaches gdb instances to running MPI processes to provide debugging capabilities. These tools allow the users to step through MPI programs as they would with a normal C/C++ program. The users are provided with a host of useful debugging tools such as breakpoints insertion, procedures stepping, viewing the values of a variable across multiple processes, and obtaining stack traces. However, like gdb, they do not provide any correctness checking and only serve as debugging IDEs. As these

tools do not require the recompilation or relinking of the source code, they are usually the only choices available if the user does not have access to the source.

6.1.2 Correctness Checking

MPI correctness checking tools are those that run the MPI programs and check for runtime errors that occur during that run. This is usually accomplished by recompilation and/or relinking the program. Since these tools are typically not aware of the alternative outcomes due to MPI nondeterminism, their abilities to detect MPI errors heavily depend on the errors that actually occur with a test harness. In other words, nondeterminism induced bugs will still pose a challenge for these tools. One standard approach is to run the program with the same test harness as many times as the computing resources permit. Unfortunately, studies have shown this technique to be rather ineffective [73]. The above study also shows that random delay might help in the case of nondeterminism, but coverage is not guaranteed, however.

To the best of our knowledge, these are currently the only MPI correctness tool available:

- Umpire [68], developed at the Lawrence Livermore National Laboratory (LLNL) by Jeffrey Vetter and Bronis de Supinski, is one of the first correctness checking tools for MPI. Despite not being actively maintained, Umpire remains a useful tool for many MPI programmers. Umpire does not require recompilation of the MPI programs being checked, but it does require relinking the MPI programs with its MPI profiling interface. At runtime, each MPI program launches several threads that communicate with the Umpire manager thread about the processes' MPI activities. The communication between the manager and the error checking threads rely on MPI itself, which means Umpire requires the MPI runtime to support `MPI_THREAD_MULTIPLE`. Umpire separates MPI error checks into local checks and global checks in which local checks include unfinished communication requests, unfreed communicators, uncommitted types, and bad arguments while global checks include deadlocks and type mismatche. In the most widely available Umpire version, deadlock checking is done through a simple dependency graph mechanism. A new deadlock detection mechanism that is based on Wait-For-Graphs and provides better scalability has been implemented as an experimental project for Umpire [32].
- MPI-CHECK [41] works only with MPI Fortran 90 programs. An experimental

C/C++ version exists but is no longer in active development. Unlike Umpire, MPI-CHECK does not rely on the MPI profiling interface. Instead, MPI-CHECK instruments the source code of the program to replace MPI calls with MPI-CHECK's own versions. During the parsing of the source code, MPI-CHECK also checks the program for usage errors (e.g., using a negative number for the destination field in `MPI_Send`). During execution, the MPI processes sends information of the execution to a centralized manager through the use of TCP sockets. MPI-CHECK can detect common errors such as deadlock, type mismatches between sends and receives, and under-allocated message buffer).

- Marmot [35,36] is an MPI checker that offers similar functionalities to Umpire. The checker uses the MPI profiling interface to intercept MPI calls and analyze them at runtime. The error checking consists of local checks and global checks, similarly to those of Umpire. Each processes handles the local checks such as resource leaks and passes along the data to a debug server, which is a separate MPI process (Marmot requires one extra process to run the debug server), for global error checking such as deadlocks. In contrast with the previous tools, Marmot uses a simple timeout-based deadlock detection scheme that has low overhead but can potentially produce false alarms. Marmot has extensive integration capabilities with other GUI tools to help the user visualize the checking results. Currently, Marmot has integrations with the following tools: Cube [6], DDT, Microsoft Visual Studio [8], Eclipse [3], and Vampir. In addition, Marmot can also detect a small number of OpenMP usage errors.
- MPIDD [30] only offers deadlock detection capabilities. It uses a centralized approach in which a separate MPI process acts as a manager and communicates with other processes through TCP socket calls and builds a dependency graph based on the data that it receives from the processes. The tool uses a standard Depth-First-Search cycle detection algorithm to detect deadlock during runtime.
- MPIRace-Check [48] is an MPI checker that focuses on message race detection for MPI programs with nondeterministic receives. This is similar to DAMPI's ability to detect all possible outcomes of nondeterministic receives. However, unlike DAMPI, MPIRace-check does not include any mechanism to replay the execution to cover the detected races. MPIRace-check uses a version of eager vector clocks discussed earlier in Section 3.2 as their central algorithm to detect message races. We have

shown earlier that this algorithm terminates the effect of nonblocking wildcard receives too early and leads to omissions. MPIRace-check also does not have the required scalability and robustness to handle large MPI programs as of this writing.

- The Intel Message Checker (IMC) [23] is an MPI checker that provides postmortem analysis of the errors that it detected during program execution. IMC has three main components: the Trace Collector, which intercepts MPI calls using the standard MPI profiling interface to collect information such as input parameters and message buffer checksum; the Analyzer Engine, which reads the trace files from the Trace Collector and analyzes them for MPI errors; and finally the Visualizer, which interprets the output from the Analyzer and allows the user to navigate to the errors. IMC detects common MPI errors such as deadlock, unsafe buffer access (i.e., accessing the buffer of a pending communication request), and type mismatches. The major drawback of this approach is that if a critical MPI error occurs and the program crashes, the behavior of the Trace Collector is undefined, which means the user might not get the trace files.
- The Intel Trace Analyzer and Collector (TAC) [47] is built on top of IMC and designed to work with Intel MPI. Unlike IMC, TAC does not rely on postmortem analysis. Similarly to Marmot and Umpire, TAC distinguishes between local checks and global checks. The local checks return not only the line number in the source code but also provide a full stack trace. In contrast with how most tools handle global checks, TAC handles global checks in a distributed fashion and does not rely on a centralized approach. Instead, each process creates different TCP-based communication channels with all other processes and communicates with them through a predefined API. This mechanism allows TAC to detect deadlock as well as type mismatches. However, this independent communication layer potentially limits the scalability of the tool.

6.2 Verification Tools

To the best of our knowledge, ISP and DAMPI are the only two tools that offer verification coverage for MPI programs over the space of nondeterminism. In other words, these are the only tools that explore the possible executions of MPI applications to detect MPI errors. Both tools also offer heuristics to limit the search space.

MPI-SPIN [56, 57] is a model checker based on SPIN that exhaustively explores all

interleavings of a nondeterministic MPI programs and verifies it for common MPI errors. However, being a traditional model checker, MPI-SPIN operates on a user-built model of the MPI program instead of the application itself. Thus, it severely restricts the applicability of the tools to small MPI programs. The vast size of practical MPI programs and their complexity make the task of manual model building impractical.

Outside of the MPI application domain, many verification tools exist for multithreaded program such as Inspect [75,76] and CHESS [7]. Inspect systematically explores different interleavings of a C multithreaded program and verifies the program for concurrency bugs such as data races and deadlocks. Inspect relies on dynamic partial order reduction as well as symmetry to reduce the number of interleavings. CHESS is another multithreaded verifier that works for .NET code. Similarly to Inspect, CHESS uses a scheduler-based approach to explore different interleavings. However, the CHESS scheduler employs a large number of search strategies aiming at finding bugs within the least number of executions. One such strategy is the context bounded search approach that bounds the number of preemptions that the scheduler allows. Our bounded mixing search strategy is inspired by this scheme. However, in the multithreaded space, this approach turns out to be very powerful since most of the concurrency bugs can be caught with a very low number of preemptions.

6.3 Deterministic Replay

The difficulty of debugging programs under the presence of nondeterminism has triggered many research efforts in deterministic replay, not just for multithreaded programs but for MPI programs as well. Deterministic replay, when used together with a parallel debugger, facilitates the process of bug tracking. In addition to debugging, deterministic replay has many other usages including fault tolerance, performance analysis, and intrusion detection.

For MPI applications, deterministic replay is a straightforward process where the MPI process writes the information of the current run, including outcomes of nondeterministic receives, to a trace file and a replay engine uses that to replay the program. In practice, the amount of logged data varies between the different tools and is highly dependent on their approaches. Data-replay tools [19,21] do not require all processes to participate in the replay since the trace files contain all data necessary to replay. This approach is useful in situations in which computing resources are expensive, which makes the requirement

of having all processes available for replaying infeasible. The major drawback of this approach is that the trace data could be very large, especially for programs that exchange huge amounts of data. Order-replay tools [37, 38] address the problem of large trace files by logging only the control information related to each message (e.g., for an `MPI_Send`, it would log the destination process, the message checksum, the number of elements, the tag, the communicator). However, all processes must participate in the replaying phase. MPIWiz [74] offers a compromise between order-replay and data-replay by grouping MPI processes into subgroups and only recording the data exchange between the groups.

Deterministic replay becomes a lot more complicated for multithreaded programs because it requires tracking read and write accesses to all shared variables as well as enforcing the same order of accesses by the threads during replays. Since tracking every access to shared data is expensive, many approaches rely on hardware modifications. The Rerun [33] tool, for example, requires several hardware counters to record reads, writes, and Lamport timestamps. The Lamport timestamps allow Rerun to correctly enforce the thread access order during replay.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The MPI standard offers a rich set of features such as nonblocking primitives and nondeterministic constructs that help developers write better high performance applications. These features, however, complicate the task of large-scale debugging, especially over the space of nondeterminism, which requires causality tracking. Traditional causality tracking algorithms, such as Lamport clocks and vector clocks, are usually not sufficient to handle such complex semantics. In this dissertation, we investigate the insufficiency of the Lamport happens-before order and propose the distributed MPI matches-before order which provides the basis for new distributed algorithms that can correctly track causality in MPI executions. To this end, we provide two logical clock protocols that can realize and maintain matches-before order among MPI events in an MPI execution. The first protocol is the Lazy Lamport Clocks Protocol (LLCP), which provides very good scalability with a small possibility of having omissions. The second protocol is the Lazy Vector Clocks Protocol (LVCP), which provides full coverage guarantee at the cost of a scalability tradeoff. In practice, we show through our experiments that LLCP provides the same coverage as LVCP.

Both protocols are implemented in our tool Distributed Analyzer for MPI (DAMPI). DAMPI implements many correctness checking modules for MPI programs and is driven by either LLCP or LVCP to provide correctness checking over the space of nondeterminism. To reduce the verification time for large programs further, we implement several heuristics that allow the user to tune the coverage to regions of interest. We evaluate DAMPI against ISP, another dynamic verifier for MPI that implements a centralized scheme of MPI matches-before, using large MPI programs such as the SpecMPI2007 benchmarks and the NAS PB benchmarks. The evaluation shows that DAMPI provides scalable and modular verification for large scale MPI programs.

7.1 Future Research Directions

7.1.1 Static Analysis Support

While search bounding heuristics are useful in reducing the number of interleavings explored by dynamic schedulers, they are still restricted by the information obtained during runtime. For example, given a trace from process P_0 showing five successive wildcard receives, it is difficult to distinguish between a wildcard receive invoked five times in five different procedures and a wildcard receive being invoked five times within a loop. Our loop abstraction iteration heuristics allow the user to exclude certain loops from the verification, but we rely on the assumption that the user has enough knowledge of the code to accomplish this task. Static analysis might automate this process and in some cases, might even detect several common errors such as buffer reuse or type mismatches before dynamic analysis takes place.

7.1.2 Hybrid Programming Support

Supercomputers with multicore nodes are becoming increasingly popular since we are likely at the limit of our technological capabilities to increase the computing power of any one processor. In fact, none of the big CPU vendors have any single core processor on their roadmaps for the foreseeable future. While recent MPI implementations do exploit multicore chips by using shared memory as their communication channel for processes mapped to different cores on the same processor, the performance is still nowhere close to multithreading. In order to exploit the computing power of multicore chips, programmers must rely on threads, whether explicitly through pthreads or OpenMP, or implicitly through libraries such as TBB. Threading and message passing create debugging challenges that are even more difficult than the scenarios that we have described. In addition to MPI-related bugs, we now also must consider data races caused by threads and the possibilities of threads making MPI calls. Even if we assume threads to not make MPI calls and focus our effort only on MPI related bugs, the challenge of ensuring the same order of threads interaction remains so that we can get the correct replay up to the point where we want to enforce the alternative choice of an MPI nondeterministic event, which likely would require the cooperation of both an MPI scheduler and a thread scheduler. With the promising future of hybrid programming and the diversity of hybrid programming models, research into extending DAMPI to handle hybrid programs will be a valuable contribution.

REFERENCES

[1] Allinea DDT. http://www.allinea.com/products/ddt/.

[2] Bounded Model Checking for ANSI-C. http://www.cprover.org/cbmc/.

[3] Eclipse. http://www.eclipse.org/.

[4] The GNU Debugger. http://www.gnu.org/software/gdb/.

[5] ISP Test webpage. http://www.cs.utah.edu/formal_verification/ISP_Tests/.

[6] Kojack CUBE. http://icl.cs.utk.edu/kojak/cube.

[7] Microsoft CHESS. http://research.microsoft.com/en-us/projects/chess/.

[8] Microsoft Visual Studio. http://www.microsoft.com/visualstudio/en-us/.

[9] MPI-2 over InfiniBand. http://mvapich.cse.ohio-state.edu/.

[10] The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[11] OpenMPI: Open Source High Performance Computing. http://www.openmpi.org/.

[12] ParMETIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

[13] The Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks.

[14] The SPECMPI2007 Benchmarks. http://www.spec.org/mpi.

[15] The ASCI Purple Benchmark. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/.

[16] Top500 Supercomputing Sites. http://www.top500.org.

[17] TotalView Software. http://www.roguewave.com/products/totalview.

[18] Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., de Supinski, B., and Schulz, M. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 368–377.

[19] Bouteiller, A., Bosilca, G., and Dongarra, J. Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds., vol. 4757 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 297–306.

[20] Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Information Processing Letters 39* (July 1991), 11–16.

[21] Clémençon, C., Fritscher, J., Meehan, M. J., and Rühl, R. An implementation of race detection and deterministic replay with MPI. In *Proceedings of the First International Euro-Par Conference on Parallel Processing* (London, UK, 1995), Euro-Par '95, Springer-Verlag, pp. 155–166.

[22] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems—Concepts and Design, 2nd Ed.* Addison-Wesley Publishers Ltd., 1994, ch. 17, pp. 517–544.

[23] DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., and Bratanov, S. Automated, scalable debugging of MPI programs with Intel® message checker. In *International Workshop on Software Engineering for High Performance Computing Applications (SE-HPCS)* (2005), pp. 78–82.

[24] Fidge, C. J. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference* (St Lucia, Australia, 1988), ACSC '88, University of Queensland, pp. 56–66.

[25] Fuller, S. H., and Millett, L. I. *The Future of Computing Performance: Game Over or Next Level?* The National Academy Press, Washington, DC, USA, 2011.

[26] Godefroid, P., Hanmer, B., and Jagadeesan, L. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal 3*, 2 (April-June 1998).

[27] Gottbrath, C. Eliminating parallel application memory bugs with TotalView. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[28] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (Sept. 1996), 789–828.

[29] Gropp, W. D., and Lusk, E. *User's Guide for MPICH, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[30] Haque, W. Concurrent deadlock detection in parallel programs. *International Journal in Computer Applications 28* (January 2006), 19–25.

[31] Hélary, J.-M., Raynal, M., Melideo, G., and Baldoni, R. Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering 15* (September 2003), 1239–1250.

[32] Hilbrich, T., de Supinski, B. R., Schulz, M., and Müller, M. S. A graph based approach for MPI deadlock detection. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 296–305.

[33] Hower, D. R., and Hill, M. D. Rerun: Exploiting episodes for lightweight memory race recording. *SIGARCH Computer Architecture News 36* (June 2008), 265–276.

[34] Humphrey, A., Derrick, C., Gopalakrishnan, G., and Tibbitts, B. GEM: Graphical explorer of MPI programs. In *39th International Conference on Parallel Processing Workshops* (September 2010), ICPPW '10, pp. 161 –168.

[35] Krammer, B., Bidmon, K., Müller, M., and Resch, M. Marmot: An MPI analysis and checking tool. In *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, F. P. G.R. Joubert, W.E. Nagel and W. Walter, Eds., vol. 13 of *Advances in Parallel Computing*. North-Holland, 2004, pp. 493 – 500.

[36] Krammer, B., and Resch, M. M. Correctness checking of MPI one-sided communication using Marmot. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI), LNCS 4192* (2006), pp. 105–114.

[37] Kranzlmüller, D., Löberbauer, M., Maurer, M., Schaubschläger, C., and Volkert, J. Automatic testing of nondeterministic parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 2* (Las Vegas, NV, USA, 2002), PDPTA '02, CSREA Press, pp. 538–544.

[38] Kranzlmüller, D., Schaubschläger, C., and Volkert, J. An integrated Record&Replay mechanism for nondeterministic message passing programs. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (London, UK, 2001), Springer-Verlag, pp. 192–200.

[39] Kranzlmüller, D., and Volkert, J. NOPE: A nondeterministic program evaluator. In *Proceedings of the 4th International ACPC Conference Including Special Tracks on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia: Parallel Computation* (London, UK, 1999), ParNum '99, Springer-Verlag, pp. 490–499.

[40] Lamport, L. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[41] Luecke, G., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., and Zou, Y. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience 15* (2003), 93–100.

[42] Lusk, R., Pieper, S., Butler, R., and Chan, A. Asynchronous dynamic load balancing. `http://www.cs.mtsu.edu/~rbutler/adlb/`.

[43] Mattern, F. Virtual time and global states of distributed systems. In *Proceedings Workshop on Parallel and Distributed Algorithms* (North-Holland / Elsevier, 1989), pp. 215–226.

[44] Meldal, S., Sankar, S., and Vera, J. Exploiting locality in maintaining potential causality. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1991), PODC '91, ACM, pp. 231–239.

[45] Musuvathi, M., and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. *ACM SIGPLAN Notices 42*, 6 (2007), 446–455.

[46] Musuvathi, M., and Qadeer, S. Fair stateless model checking. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), ACM, pp. 362–371.

[47] Ohly, P., and Krotz-vogel, W. Automated MPI correctness checking What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing* (South Lake Tahoe, CA, USA, May 2007), LCI'07, pp. 19–25.

[48] Park, M.-Y., Shim, S., Jun, Y.-K., and Park, H.-R. MPIRace-Check: Detection of message races inMPIprograms. In *Advances in Grid and Pervasive Computing*, vol. 4459 of *Lecture Notes in Computer Science*. 2007, pp. 322–333.

[49] Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., and Kline, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering 9* (May 1983), 240–247.

[50] Schulz, M. Extracting critical path graphs from MPI applications. In *Cluster Computing, 2005. IEEE International* (Boston, MA, USA, September 2005), pp. 1 –10.

[51] Schulz, M., Bronevetsky, G., and de Supinski, B. R. On the performance of transparent MPI piggyback messages. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 194–201.

[52] Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., and Stodghill, P. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme forMPIprograms. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2004), SC '04, IEEE Computer Society, pp. 38–.

[53] Schulz, M., and de Supinski, B. R. P^nMPI tools: A whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2007), SC '07, ACM, pp. 30:1–30:10.

[54] Shende, S., Malony, A. D., Morris, A., and Wolf, F. Performance profiling overhead compensation for MPI programs. In *PVM/MPI* (2005), B. D. Martino, D. Kranzlmüller, and J. Dongarra, Eds., vol. 3666 of *Lecture Notes in Computer Science*, Springer, pp. 359–367.

[55] Shende, S. S., and Malony, A. D. The Tau parallel performance system. *International Journal on High Performance Computer Applications 20* (May 2006), 287–311.

[56] Siegel, S. F. MPI-Spin web page. `http://vsl.cis.udel.edu/mpi-spin`, 2008.

[57] Siegel, S. F., and Avrunin, G. Verification of MPI-based software for scientific computation. In *International SPIN Workshop on Model Checking Software* (Apr. 2004), pp. 286–303.

[58] Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17–20, 2006* (2006), L. L. Pollock and M. Pezzé, Eds., ACM, pp. 157–168.

[59] Siegel, S. F., and Siegel, A. R. MADRE: The Memory-Aware Data Redistribution Engine. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings* (2008), A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds., vol. 5205 of *LNCS*, Springer.

[60] Singhal, M., and Kshemkalyani, A. An efficient implementation of vector clocks. *Information Processing Letters 43*, 1 (1992), 47 – 52.

[61] Sun, Y., Lin, X., Ling, Y., and Li, K. Broadcast on clusters of SMPs with optimal concurrency. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 4* (Las Vegas, NV, USA, 2002), PDPTA '02, CSREA Press, pp. 1558–1564.

[62] Torres-Rojas, F. J., and Ahamad, M. Plausible clocks: Constant size logical clocks for distributed systems. In *Proceedings of the 10th International Workshop on Distributed Algorithms* (London, UK, 1996), Springer-Verlag, pp. 71–88.

[63] Vakkalanka, S. *Efficient dynamic verification algorithms for MPI applications.* PhD thesis, University of Utah, Salt Lake City, Ut, USA", 2010.

[64] Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., Thakur, R., and Gropp, W. Implementing efficient dynamic formal verification methods for MPI programs. In *EuroPVM/MPI* (2008).

[65] Vakkalanka, S., Gopalakrishnan, G., and Kirby, R. M. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2008), CAV '08, Springer-Verlag, pp. 66–79.

[66] Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. Reduced execution semantics of MPI: From theory to practice. In *International Symposium on Formal Methods (FM)* (2009), pp. 724–740.

[67] Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent advances in the Message Passing Interface* (Berlin, Heidelberg, 2010), EuroMPI'10, Springer-Verlag, pp. 152–159.

[68] Vetter, J. S., and de Supinski, B. R. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Conference on*

Supercomputing (CDROM) (Washington, DC, USA, 2000), Supercomputing '00, IEEE Computer Society.

[69] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B. R. d., Schulz, M., and Bronevetsky, G. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–10.

[70] Vo, A., and Gopalakrishnan, G. Scalable verification of MPI programs. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2019, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings* (2010), IEEE, pp. 1–4.

[71] Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Formal verification of practical MPI programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 261–270.

[72] Vo, A., Vakkalanka, S. S., Williams, J., Gopalakrishnan, G., Kirby, R. M., and Thakur, R. Sound and efficient dynamic verification of MPI programs with probe non-determinism. In *EuroPVM/MPI* (2009), pp. 271–281.

[73] Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., and Sornsen, A. Improving distributed memory applications testing by message perturbation. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging* (New York, NY, USA, 2006), PADTAD '06, ACM, pp. 27–36.

[74] Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., and Voelker, G. MPIWiz: Subgroup reproducible replay of MPI applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 251–260.

[75] Yang, Y., Chen, X., Gopalakrishnan, G., and Kirby, R. M. Distributed dynamic partial order reduction based verification of threaded software. In *Proceedings of the 14th International SPIN Conference on Model Checking Software* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 58–75.

[76] Yang, Y., Chen, X., Gopalakrishnan, G., and Wang, C. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 279–295.